FLASH: A Framework for Programming Distributed Graph Processing Algorithms

Xue Li[1], Ke Meng[1], Lu Qin[2], Longbin Lai[1], Wenyuan Yu[1], Zhengping Qian[1], Xuemin Lin[3], Jingren Zhou[1]
[1]Alibaba Group
[2]Centre for Artificial Intelligence, University of Technology, Sydney, Australia
[3]Antai College of Economics & Management, Shanghai Jiao Tong University
[1]{youli.lx, mengke.mk, longbin.lailb, wenyuan.ywy, zhengping.qzp, jingren.zhou}@alibaba-inc.com
[2]lu.qin@uts.edu.au [3]xuemin.lin@gmail.com

***Abstract*—As a result of decades of studies, a broad spectrum of graph algorithms have been developed for graph analytics, including clustering, centrality, traversal, matching, mining, etc. However, the majority of recent graph processing frameworks only focus on a handful of fix-point graph algorithms such as breadth-first search, PageRank, shortest path, etc. It leaves the distributed computation of a large variety of graph algorithms suffering from low efficiency, limited expressiveness, or high implementation complexity with existing frameworks.**

In this paper, we propose FLASH, a framework for programming distributed graph processing algorithms, which achieves good expressiveness, productivity and efficiency at the same time. Thanks to its high-level interface, FLASH allows users to implement complex distributed graph algorithms with high performance with only a few lines of code. We have implemented 72 graph algorithms for 49 different problems in FLASH. In further evaluations, we found that FLASH beats other state-of-the-art graph processing frameworks with the speedups of up to 2 orders of magnitudes while takes up to 92% less lines of code.

***Index Terms*—Graph computing, Programming model, Distributed computing, Code generation**

I. INTRODUCTION

Graph algorithms serve as the essential building blocks for a diverse variety of real-world applications such as social network analytics [2], data mining [3], network routing [4], and scientific computing [5]. As the graph data becomes increasingly huge, there is an urgent need of scaling graph algorithms in the distributed context, and many distributed graph frameworks have emerged to fill in the gap, such as Pregel [6], Giraph [7], GraphLab [8], PowerGraph [9], GraphX [10], Gemini [11] and others [12], [13], [14], [15], [16]. However, as pointed out by [17], all these graph processing frameworks are only evaluated via a handful of specific graph algorithms that are similar in computation patterns. Such evaluation is far from sufficient lacking in the coverage of diversity and usability required from real-world applications. To comprehensively evaluate a distributed graph framework, we carefully consider three metrics that we call **EPE**, namely *expressiveness*, *productivity* and *efficiency*. Specifically,

- Expressiveness represents the capability of the programming interface to express different kinds of graph algorithms. Expressiveness is arguably the most important metric that must be fulfilled in order to support the diverse graph applications in practice.
- Productivity shows the convenience of the interface or the required effort for users to implement their algorithms, which is also important given that otherwise graph computation will be a privilege to a few expertise [16].
- Efficiency refers to the performance factor of executing graph algorithms, which should be concerned because real-world applications on large-scale graphs are often time-consuming and/or memory-intensive.

After thoroughly analyzing existing graph processing frameworks, we find out that none of them has arrived at the proper tradeoff for all the EPE metrics. To pursuit productivity, an abstraction called "thinking like a vertex" (or vertex-centric) was proposed in Pregel [6], and shared among many existing graph processing frameworks. Under this philosophy, graphs are divided into partitions for scalability and updates are bound to the granularity of vertices for parallelization. The vertex-centric implementation of a graph algorithm follows a common iterative, single-phased and value-propagation-based (short of *ISVP*) pattern [17]: the algorithm runs iteratively until convergence, and in each iteration, all vertices receive messages from their neighbors to update their own states, then they send the updated states as messages to the neighbors for the next iteration. Due to the productivity of the vertex-centric model, a lot of graph frameworks follow the philosophy for their abstraction, including the GraphLab variant [8], the Scatter-Gather model [18], and the *GAS* model [9].

Such high-level abstraction brings productivity to some extent to users, however, at the sacrifice of expressiveness and efficiency. Regarding expressiveness, we argue that the abstraction, while designed specifically for the ISVP algorithms, is almost infeasible to be applied to a large variety of algorithms that are not of the kind, such as K-clique Counting, Rectangle Counting, Betweenness Centrality and the optimized Connected Components algorithm [19], to just name a few. Actually, there are typically tens of algorithms available to representative graph processing frameworks. As a comparison, NetworkX [20], a Python graph library, supports over 328 graph algorithms. At the same time, modern graph scenarios bring in the needs of more advanced and complex graph algorithms, which poses a big challenge for existing graph processing frameworks. After investigating representative distributed graph algorithms, including many non-ISVP ones, we have distilled three requirements that are critical for programming them efficiently and productively in a distributed context, namely (R1) flexible control flow; (R2) operations on vertex sets; and (R3) beyond-neighborhood communication. As examples, Table I lists what requirements are needed by the evaluated algorithms (more algorithms can be found in [1]). However, existing graph frameworks all fall short in meeting these requirements. For example, Pregel does not offer R1 and R2, while GAS fails to support all three requirements. Ligra [21], [22] is the programming model that has fulfilled the most requirements so far (R1 and R2). However, Ligra still has limitations for expressing non-ISVP algorithms because of the absence of R3. In addition, it is built upon a shared-memory architecture, thus not suitable for programming graph algorithms in a distributed context.

M1, R1.O1, R3.O2

Regarding efficiency, it's not surprising that some hard-coded algorithms can run much faster than the high-level alternative implementation on a framework. The general graph frameworks, especially distributed frameworks lose efficiency because of communication/synchronization overhead, load balance issues and higher software complexities [11]. As we evaluated, a hardcoded algorithm for Connected Components [19] can perform orders of magnitude faster than the vertex-centric counterparts. To overturn the efficiency issue, Gemini [11] has been developed that significantly lower the latency of executing certain algorithms. However, the user is often required to program more codes compared to the vertex-centric alternatives, which harms the productivity to some degree. Moreover, in order to exploit the extreme efficiency, some constraints are enforced by Gemini that can deteriorate the expressiveness: it does not support the above three critical requirements, and it requires the vertex property to be fixed-length and the core computation to satisfy the *associative* and *commutative* properties. Obviously, a graph framework based on the low-level interfaces such as MPI can provide the optimal result for expressiveness and efficiency, but it may not be appealing to the users in terms of productivity.

Motivated by this, we propose a new distributed graph processing framework called FLASH in this paper, and claim that FLASH finally approaches a sweet spot for all EPE metrics. We base the FLASH programming model on Ligra to inherit its support for the requirements of flexible control flow and operations on vertex sets. By further enabling beyond-neighborhood communication, FLASH improves the expressiveness for programming a diverse variety of distributed graph algorithms. Note that Ligra is a single-machine parallel library, and our extension of Ligra to the distributed context is non-trivial, for which we must carefully handle communication, synchronization, data races and task scheduling. To do so, we propose a middleware called FLASHWARE that hides all the above details for distribution, and provides the capability to apply multiple system optimizations automatically and adaptively at the runtime. In fact, we have implemented 72 graph algorithms in FLASH for 49 different commonly used applications. Among them, some algorithms are extremely

M1, R1.O1

TABLE I: Comparison of different models.

Algo. [20]	**Pregel**	**GAS**	**Gemini**	**Ligra**	**FLASH**
Expressiveness & Productivity [LLoCs [27], lower is better]					
CC-basic	● 30	● 36	● 50	● 26	● 12
CC-opt[13]	◒ 63	○	○	○	● 56
BFS	● 22	● 25	● 56	● 20	● 13
BC[12]	◒ 49	◒ 162	◒ 139	● 75	● 33
MIS[2]	● 48	● 53	● 112	● 37	● 23
MM-basic	● 57	● 66	● 98	● 59	● 20
MM-opt[13]	◒ 84	○	○	○	● 27
KC[1]	◒ 35	◒ 32	○	● 45	● 20
TC	● 31	◒ 181	○	● 38	● 22
GC	● 48	● 58	○	○	● 24
SCC[12]	◒ 275	○	○	○	● 74
BCC[12]	◒ 1057	○	○	○	● 77
LPA	● 51	● 46	○	○	● 26
MSF[13]	◒ 208	○	○	○	● 24

The algorithms are demonstrated in the full version [1].
The three critical requirements are: flexible control flow[1], operations on vertex sets[2] and beyond-neighborhood communication[3].
● means that it is well-supported; ○ means that we failed to express it; ◒ means that it could be implemented in a non-intuitively way at the cost of performance.

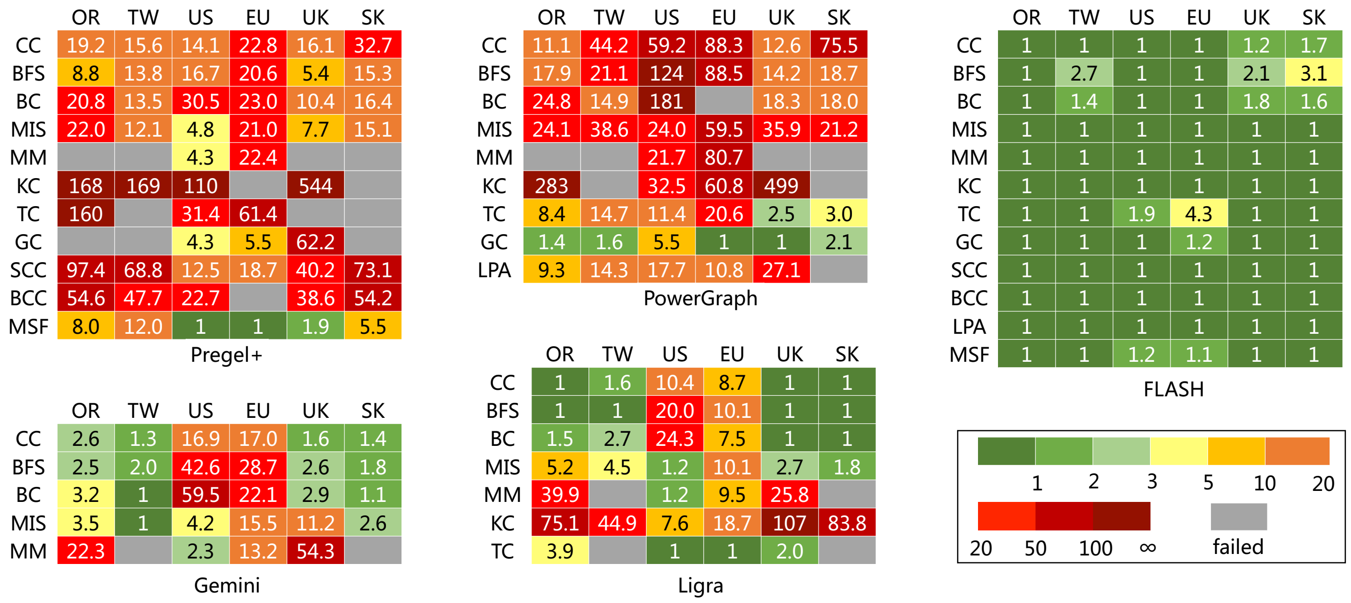
difficult or nearly impossible to implement on existing graph processing frameworks, including the optimized algorithm to compute minimum spanning tree [19], faster betweenness centrality algorithm [23], k-core decomposition [24], graph coloring [25], k-clique counting [26], to just name a few. Moreover, thanks to FLASHWARE, we can now program much more succinct codes using the FLASH programming interfaces, which also helps productivity. Table I summarizes the comparison in expressiveness and productivity of FLASH and the representative graph frameworks, while Figure 1 presents the comparison in efficiency. No a single framework can beat FLASH in all metrics, while FLASH does outperform Pregel+ and PowerGraph by a large margin in all aspects.

In summary, we make the following contributions.

(1). We propose the FLASH model, a novel high-level abstraction for programming distributed graph algorithms (Section III). We define its succinct interface after presenting some preliminaries (Section II), and show its expressiveness through some representative examples. We also make a comparison with other models to highlight FLASH's advantages.

(2). We provide an efficient implementation of FLASH (Section IV) based on a novel design of the system architecture and a middleware named FLASHWARE. Moreover, we explore adaptive runtime choices to further enhance the performance.

(3). We provide a thorough experimental evaluation of FLASH considering the EPE metrics (Section V). The results demonstrate FLASH's capability of expressing many advanced algorithms, while providing a satisfactory performance at the same time. It beats other state-of-the-art frameworks in 84.5% cases, and may achieve significant speedups of up to 2 orders of magnitudes while takes up to 92% less lines of code.


1
2
3
5
10
20
20
50
100
∞
failed

Pregel+

	OR	TW	US	EU	UK	SK
CC	19.2	15.6	14.1	22.8	16.1	32.7
BFS	8.8	13.8	16.7	20.6	5.4	15.3
BC	20.8	13.5	30.5	23.0	10.4	16.4
MIS	22.0	12.1	4.8	21.0	7.7	15.1
MM	▨	▨	4.3	22.4	▨	▨
KC	168	169	110	▨	544	▨
TC	160	▨	31.4	61.4	▨	▨
GC	▨	▨	4.3	5.5	62.2	▨
SCC	97.4	68.8	12.5	18.7	40.2	73.1
BCC	54.6	47.7	22.7	▨	38.6	54.2
MSF	8.0	12.0	1	1	1.9	5.5

PowerGraph

	OR	TW	US	EU	UK	SK
CC	11.1	44.2	59.2	88.3	12.6	75.5
BFS	17.9	21.1	124	88.5	14.2	18.7
BC	24.8	14.9	181	▨	18.3	18.0
MIS	24.1	38.6	24.0	59.5	35.9	21.2
MM	▨	▨	21.7	80.7	▨	▨
KC	283	▨	32.5	60.8	499	▨
TC	8.4	14.7	11.4	20.6	2.5	3.0
GC	1.4	1.6	5.5	1	1	2.1
LPA	9.3	14.3	17.7	10.8	27.1	▨

FLASH

	OR	TW	US	EU	UK	SK
CC	1	1	1	1	1.2	1.7
BFS	1	2.7	1	1	2.1	3.1
BC	1	1.4	1	1	1.8	1.6
MIS	1	1	1	1	1	1
MM	1	1	1	1	1	1
KC	1	1	1	1	1	1
TC	1	1	1.9	4.3	1	1
GC	1	1	1	1.2	1	1
SCC	1	1	1	1	1	1
BCC	1	1	1	1	1	1
LPA	1	1	1	1	1	1
MSF	1	1	1.2	1.1	1	1

Gemini

	OR	TW	US	EU	UK	SK
CC	2.6	1.3	16.9	17.0	1.6	1.4
BFS	2.5	2.0	42.6	28.7	2.6	1.8
BC	3.2	1	59.5	22.1	2.9	1.1
MIS	3.5	1	4.2	15.5	11.2	2.6
MM	22.3	▨	2.3	13.2	54.3	▨

Ligra

	OR	TW	US	EU	UK	SK
CC	1	1.6	10.4	8.7	1	1
BFS	1	1	20.0	10.1	1	1
BC	1.5	2.7	24.3	7.5	1	1
MIS	5.2	4.5	1.2	10.1	2.7	1.8
MM	39.9	▨	1.2	9.5	25.8	▨
KC	75.1	44.9	7.6	18.7	107	83.8
TC	3.9	▨	1	1	2.0	▨

Fig. 1: A heat map of slowdowns of various frameworks compared to the fastest framework for 12 representative applications on 6 real-world graphs. "failed" means that the execution did not terminate within 5000s or failed due to exhausted memory.

II. Preliminaries and Related Work

We first define some basic graph notations and discuss the mainstream programming models of existing frameworks.

Functions. FLASH uses the functional programming paradigm. A variable with a type is denoted as $var : type$. A function $f(x : X) \mapsto Y$ means that for each input $x \in X$, it has an output $y \in Y$ such that $f(x) = y$.

Graphs. FLASH is defined over the property graph, which can be represented as $G = (V, E)$. V represents a finite set of vertices, and each $v \in V$ has a unique identifier ($v.id \in \mathbb{N}$) and some associated properties. With the cartesian product of sets A and B denoted by $A \times B$ where $A \times B = \{(a, b) : a \in A \wedge b \in B\}$, $E \subseteq V \times V$ represents a set of (directed) edges. For each edge (s, d), the first vertex s is the source of it and the second vertex d is the target. The number of vertices in a graph is denoted by $|V|$ and the number of edges is denoted by $|E|$. We denote a weighted graph by $G = (V, E, w)$, where w is a function which maps an edge to a real value, thus each edge $e \in E$ is associated with the weight $w(e)$.

Graphs partitions. In a distributed cluster containing m workers, the input graph will be partitioned into m partitions (also called subgraphs), with each worker holding one of the partitions ($P_i = (V_i, E_i)$ for worker i). A m-way partition scheme for the graph $G(V, E)$ should guarantee that $V = \cup_{i=1}^{m} V_i$ and $E = \cup_{i=1}^{m} E_i$. There are two main schemes for partitioning the graph, namely *edge-cut* and *vertex-cut*, and we use *edge-cut* which is more common in existing works. To R3.O4 be specific, each vertex belongs to only one partition, so that $V_i \cap V_j = \emptyset$, for $i \neq j$. Two endpoints of an edge may belong to different partitions. To this end, FLASH uses the master-mirror notion as PowerGraph [9] proposed: each vertex is assigned to and owned by one partition, where it is a *master*, as its primary copy. While in other partitions, there may also be replicas for this vertex, called *mirrors*. We will explain this more specifically in Section IV.

Graph algorithms. A graph algorithm takes a graph G as input, does processing and analytics on it to solve real-world problems. As assumed in many works, a graph algorithm updates the information stored in vertices, while edges are viewed as immutable objects. Generally, A graph algorithm propagates updates along the edges (original edges of G, or virtual edges generated dynamically during the execution) iteratively, until the convergence condition is met or a given number of iterations are completed. Vertices with ongoing updates are called *active vertices* (or *frontiers*), with outgoing edges from them called *active edges*. This paper considers the Bulk Synchronous Parallel (BSP) model [28], where an algorithm consists of a series of supersteps. In each superstep, computation and communication (message passing) take place on active vertices. The supersteps are executed synchronously, so that messages sent during one superstep are guaranteed to be delivered in the beginning of the next superstep.

Pregel. There are many parallel programming abstractions for processing large graphs [29], [30], [17]. Pregel [6] proposed a vertex-centric abstraction which is designed based on the BSP model. By distributing vertices and associated adjacent edges to each worker, the Pregel framework has fixed routines for all the algorithms. In each superstep, a Pregel program calls the `compute()` on each vertex, which performs the pre-defined user-specific computation. During the `compute()`, each vertex processes the incoming messages from the previous superstep, then sends messages to other vertices. The communication is based on message-passing, with messages can be aggregated if the user provides a `combine()` function. The early-aggregation of messages can reduce the number of messages to be materialized, thus improves the performance and reduces the memory usage. The Pregel model greatly simplifies the parallel graph algorithms. Therefore, it almost becomes a standard of large graph processing, followed by many graph processing frameworks [7], [13], [12], [10], [11].

M5 However, the Pregel model only exposes a single vertex to the users, and thus lacks in the global perspective, for example, to maintain a group of specific vertices. Additionally, the loop control flow is *constrained* by the model, making it only suitable for single-phased algorithms. As an example, the Betweenness Centrality algorithm [23] requires both flexible control flow (e.g. recursive loop) and operating on frontiers as vertex sets in each step, which cannot be easily expressed by this model. Actually, the implementation of it provided by [11] needs more than 400 lines of code in total. Following this vertex-centric philosophy, FLASH makes an improvement on the expressiveness by allowing the users to define flexible control flow and materialize vertex sets to operate upon.

GAS. GraphLab [8] and PowerGraph [9] are also based on the vertex-centric abstraction. They proposed the "Gather-Apply-Scatter" (GAS) model to further simplify the graph algorithms, but at the cost of limiting the data-exchange to only happen between adjacent vertices. GAS hides the communication details from programmers, and the users only have the view of each vertex and its neighbors, which means that the control flow of a graph algorithm is highly rigid. Thus, GAS is less expressive but more effortless than Pregel. M5 In addition to supporting flexible control flow and operating on arbitrary vertex sets, FLASH removes the between-neighborhood limitation of GAS to support the communication beyond neighbors.

Ligra. Ligra [21] proposed a new model that supports a ***vertexSubset*** type. It represents a subset of vertices $U \subseteq V$ of the graph G. Based on this data structure, it is easy to apply the update logics on any subset of vertices every time. It exposes the control flow to the users, supporting arbitrary combination of these operations during the execution. Ligra is proved to be useful when programming a wide variety of parallel graph algorithms in the shared-memory environment.

Nevertheless, Ligra is still limited in some aspects. Since the communication is through the EDGEMAP interface, only the messages along the edges are possible. However, communication beyond neighborhood or along a set of specific edges is an important feature in some advanced graph algorithms [19]. And Ligra is designed for shared-memory systems, thus lacks the distributed semantics. It requires the programmers to use low-level instructions (e.g., the compare-and-swap instruction), which are not fitted in the distributed scenario. M5 By extending Ligra's model, FLASH is more expressive that supports communicating beyond neighbors. More importantly, FLASH does not depend on the shared-memory architecture, making it suitable for distributed processing.

III. PROGRAMMING MODEL

As depicted in Section II, previous frameworks fail to achieve the **EPE** metrics due to fixed control flow, lack of view on arbitrary vertex subsets, neighborhood-exchange limitation and the dependence on the shared-memory architecture. To address the challenges, we propose the FLASH programming model by making a sufficient extension to Ligra since Ligra addressed the first two challenges. In this section, we first show the interface of FLASH, and how to flexibly express graph algorithms with the given primitives. Then we show that our FLASH model can be fully compatible with the well-known vertex-centric models. Finally, we highlight some characteristics of FLASH to show its advantages.

A. *Interface*

FLASH is a functional programming model specific for distributed graph processing. It follows the Bulk Synchronous Parallel (BSP) computing paradigm [28], with each of the primary functions (SIZE, VERTEXMAP and EDGEMAP) constitutes a single superstep. We made the interface of FLASH much similar to Ligra's, therefore, it is easy to port a program written in Ligra to our model. The *vertexSubset* type represents a set of vertices of the graph G, which only contains a set of integers, representing the vertex id for each vertex in this set. The associated properties of vertices are maintained only once for a graph, shared by all *vertexSubset*s. The following describes the APIs of FLASH based on this type.

(1). SIZE$(U : vertexSubset) \mapsto \mathbb{N}$

This function returns the size of a *vertexSubset*, i.e., $|U|$.

(2). VERTEXMAP$(U : vertexSubset,$
$F(v : vertex) \mapsto bool,$
$M(v : vertex) \mapsto vertex) \mapsto vertexSubset$

The VERTEXMAP interface applies the map function M to each vertex in U that passes the condition checking function F. The ids of the output vertices form the resulting *vertexSubset*. That is to say, we have:

$Out = \{v.id | v.id \in U \wedge F(v) = true\}$

$v^{new} = M(v),\ v.id \in U \wedge F(v) = true$

This function is used to conduct local updates. Specially, the M function could be omitted for implementing the *filter* semantics, with the vertex data unchanged. The execution of VERTEXMAP on each vertex is independent, thus it can run in parallel naturally, as shown in Algorithm 1.

(3). EDGEMAP$(U : vertexSubset,$
$H : edgeSet,$
$F(s : vertex, d : vertex) \mapsto bool,$
$M(s : vertex, d : vertex) \mapsto vertex,$
$C(v : vertex) \mapsto bool,$
$R(t : veretex, d : vertex) \mapsto vertex) \mapsto vertexSubset$

The EDGEMAP interface in Ligra is used to transfer messages between neighbor vertices. We extend and redefine this interface for stronger expressiveness and the support for distributed semantics. For a graph $G = (V, E)$, EDGEMAP

Algorithm 1 VERTEXMAP

```
1: function VERTEXMAP(U, F, M)
2:     $Out = \{\}$
3:     parfor $u.id \in U$ do
4:         if ($F(u) == true$) then
5:             $u^{new} = M(u)$
6:             Add $u.id$ to $Out$
7:     return $Out$
```

applies the update logic to the specific edges with source vertex in U and target vertex satisfying C. H represents the edge set to conduct updates, which is E in common cases. We allow the users to define arbitrary edge sets they want dynamically at runtime, even virtual edges generated during the algorithm's execution. The edge set can be defined through defining a function which maps a source vertex id to a set of ids of the targets. We also provide some pre-defined operators for convenience, such as reverse edges (`reverse(E)`), two-hop neighbors (`join(E, E)`), or edges with targets in U (`join(E, U)`). This extension makes the communication beyond the neighborhood-exchange limitation.

If a chosen edge passes the condition checking (F), the map function M is applied on it. The output of the function M represents a temporary new value of the target vertex. This new value is applied immediately and sequentially if it is in the *pull* mode, while in the *push* mode, another parameter R is required to apply all the temporary new values on a specific vertex to get its final value. It is unnecessary in the Ligra's API because it is a shared-memory framework, which uses atomic operations to ensure consistency. On the contrary, the FLASH model is designed for distributed systems, so we use a reduce function R, which takes an old value and a new value for a single vertex, and reduces them to output the updated state. The updated target vertices form the output set of EDGEMAP. The reduce function R should be associative and commutative to ensure correctness, or R is not required for sequentially applying M, i.e., to run EDGEMAP always in the *pull* mode, as we will explained in Section III-C.

More precisely, the active edge set is defined as:
$E_a = \{(s,d) \in H | s.id \in U \wedge C(d) = true\}$.
Then, F and M are applied to each element in E_a. If it is in the *pull* mode:
$d^{new} = M(s, d^{new}), (s,d) \in E_a \wedge F(s, d^{new}) = true.$
Or, in the *push* mode:
$T = \{M(s,d) | (s,d) \in E_a \wedge F(s,d) = true\}$,
$d^{new} = R(..., R(T_2^d, R(T_1^d, d))), T_i^d \in T \wedge T_i^d.id = d.id.$
And the ids of the updated targets form output set:
$Out = \{d.id | (s,d) \in E_a \wedge F(s,d) = true\}$.

The function C is useful in algorithms where a value associated with a vertex only needs to be updated once. We retain the default function used by Ligra (CTURE) which always returns true, since the user does not need this functionality sometimes. Similarly, the F function of EDGEMAP and VERTEXMAP can also be supplied using CTURE, if it is unnecessary.

The auxiliary operators. Other auxiliary APIs are provided by FLASH for conveniently conducting set operations, including UNION, MINUS, INTERSACT, ADD, CONTAIN and so on.

B. Examples

To demonstrate the usage of FLASH and display its ability to express graph algorithms, we show two representative examples. Please refer to the full version [1] for more examples.

Breadth First Search (BFS). As with standard parallel BFS algorithms [31], [32], we implement BFS in FLASH, as shown in Algorithm 2. For each vertex, a property named dis is

Algorithm 2 BREADTH-FIRST SEARCH

```
function INIT($v, root$):
    $v.dis = (v.id == root ? 0 : INF)$
    return v
function FILTER($v, root$): return $v.id == root$
function UPDATE($s, d$):
    $d.dis = s.dis + 1$
    return $d$
function COND($v$): return $v.dis == INF$
function REDUCE($t, d$): return $t$

11: $U$ = VERTEXMAP($V$, CTRUE, INIT.bind(0))          ▷ initialize
$U$ = VERTEXMAP($V$, FILTER.bind(0))                   ▷ root=0
while SIZE($U$) $\neq 0$ do
    $U$ = EDGEMAP($U, E$, CTRUE, UPDATE, COND, REDUCE)
```

created and initialized to represent the distance from the root to this vertex. On each iteration/superstep i (starting from 0), the frontier U_i contains all vertices reachable from the root in i hops ($v.dis = i$, $\forall v \in U_i$). At the beginning of the algorithm, a *vertexSubset* that only contains the root is created, representing the frontier. To use a global variable such as r in a local function, we provide a *bind* operator to supply additional input parameters (line 11). In each of the following supersteps, the EDGEMAP function is applied on outgoing edges of the frontier, to check if any neighbor d of an active vertex s is visited. If d has not been visited, updates it and then adds it to the next frontier. The COND function tells EDGEMAP to only consider the neighbors not been visited. Although it could be replaced by CTRUE, we provide it for efficiency. As dis for a vertex d is ensured to be same no matter it is updated by which neighbor in the same superstep, we can simply remain any new value for it in the REDUCE function. The iterative execution will terminate when there are no vertices in the frontier, means that all reachable vertices from the root have been visited.

Betweenness Centrality (BC). The betweenness centrality index [33] is useful in social network analysis and has been well studied. Precisely, the betweenness centrality of a vertex v, denoted by $C_B(v)$, is equal to $\sum_{s \neq v \neq t \in V} \delta_{st}(v)$. The pair-dependency $\delta_{st}(v)$ is calculated by $\frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through v. To compute it, [23] proposed an optimized algorithm.

This algorithm works in two phases, the first phase uses a BFS-like procedure to calculate the number of shortest paths from r to each vertex, and the second phase computes the dependency scores through a backward propagation. Since the frontiers visited in every step of the first phase need to be tracked, it is difficult to directly implement this algorithm in a traditional vertex-centric model which does not supply a *vertexSubset* structure. In contrast, its implementation in FLASH is intuitively, as Algorithm 3 shows. In the second phase, the edges are need to point in the reverse direction, so we define the edge set for the EDGEMAP to be `reverse(E)`.

C. Advantages of FLASH

Expressing other programming models. FLASH has the ability to simulate the traditional vertex-centric models. So

Algorithm 3 BETWEENNESS CENTRALITY

```
1: function INIT($v, r$):
2:     if ($v.id == r$) then $v.level = 0$, $v.num = 1$, $v.b = 0$
3:     else  $v.level = -1$, $v.num = 0$, $v.b = 0$
4:     return $v$
5: function FILTER($v, r$): return $v.id == r$
6:
7: function UPDATE1($s, d$):
8:     $d.num = d.num + s.num$
9:     return $d$
10: function COND1($v$): return $v.level == -1$
11: function R1($t, d$):
12:     $d.num = d.num + t.num$
13:     return $d$
14:
15: function LOCAL($v, curLevel$):
16:     $v.level = curLevel$
17:     return $v$
18:
19: function F2($s, d$): return $d.level == s.level - 1$
20: function UPDATE2($s, d$):
21:     $d.b = d.b + \frac{d.num}{s.num} * (1 + s.b)$
22:     return $d$
23: function R2($t, d$):
24:     $d.b = d.b + t.b$
25:     return $d$
26:
27: function BC($S$, $curLevel$)
28:     if (SIZE($S$) == 0) then return
29:     $A$ = EDGEMAP ($S, E$, CTRUE, UPDATE1, COND1, R1)
30:     $A$ = VERTEXMAP ($A$, CTRUE, LOCAL.bind($curLevel$))
31:     BC($A, curLevel + 1$)
32:     $A$ = EDGEMAP ($S$, reverse($E$), F2, UPDATE2, CTRUE, R2)
33:
34: $r = 0$                                        ▷ choose a BFS root
35: $U$ = VERTEXMAP($V$, CTRUE, INIT.bind($r$))             ▷ initialize
36: $U$ = VERTEXMAP($V$, FILTER.bind($r$))          ▷ create the frontier
37: BC($U$, 1)             ▷ calculate the betweenness centrality scores
```

it is possible to port existing vertex-centric programs in our model. Below we outline a proof (see the full version [1] for details). In each superstep of the vertex-centric model, all active vertices (called *frontiers*) execute the same user-defined vertex function in parallel, which receives a set of messages as input ($inbox$) and can produce one or more messages as output ($outbox$). At the end of a superstep, the runtime receives the messages from the $outbox$ of each vertex and computes the set of active vertices for the next superstep. The local computation in each superstep can be implemented in FLASH through the VERTEXMAP, which processes the $inbox$ to produce the updated value and the $outbox$ for each vertex, and a following EDGEMAP function adds a message to the $inbox$ of the target.

Support for the three critical requirements. Besides expressing existing vertex-centric algorithms, FLASH provides the possibility of expressing more advanced algorithms. It is the first distributed graph processing model that satisfies all of the three critical requirements for programming non-ISVP algorithms. (1) We allow the users to define the arbitrary control flow by combining the primitives, thus FLASH can naturally support multi-phased algorithms. In traditional vertex-centric models, these algorithms are supported in an awkward way since they only allow to provide a single user-defined function. (2) The *vertexSubset* structure supplements the perspective of a single vertex, allowing to conduct updates on arbitrary vertices. Multiple vertex sets can be maintained at the same time, they can even be defined in a recursive function. Without this feature, a framework has to start from the whole graph every time and pick up specific vertices. (3) FLASH makes an extension to Ligra by allowing the users to provide the arbitrary edge set they want to transfer messages, even when the edges do not exist in the original graph. Therefore, algorithms that contain communication beyond neighborhood can be expressed intuitively, such as the optimized CC algorithm [19].

Dual update propagation model. During graph processing, the type of an active set may be *dense* or *sparse*, typically determined by the size of active vertices and associated outgoing edges. Ligra dispatches different computation kernels for different types of the active set: the *push* mode for sparse active sets and the *pull* mode for dense active sets. FLASH follows this design and extend it to fit the distributed scenario, using an adaptive switching between these two modes.

The EDGEMAP function calls one of EDGEMAPDENSE and EDGEMAPSPARSE (Algorithm 4 - 6) according to the density, which implements the *pull* mode and the *push* mode, respectively. The users can set the threshold to decide if it is dense. EDGEMAPDENSE loops all vertices in the graph in parallel and for each vertex v, it sequentially applies F and M for its neighbors that are in U and the edges are in H, until C returns false. After that, its id will be added to the result. Since all updates are applied immediately, R is omitted.

On the contrary, EDGEMAPSPARSE loops all vertices in the active set U in parallel and for each vertex $u \in U$, it executes $F(u, ngh)$ and $M(u, ngh)$ in parallel to update its qualified neighbors. If a neighbor is updated, it will be added to Out. As a vertex may be updated by different neighbors at the same time in a single EDGEMAPSPARSE function, all the new values will be applied on the target vertex through the R function.

This auto-switch scheme is proved to be useful for real-world graphs and is also adopted by some other works [34], [35], [11]. Also, FLASH 's dual mode processing is optional: users may choose to execute in only one mode through calling EDGEMAPDENSE/EDGEMAPSPARSE, instead of EDGEMAP.

Algorithm 4 EDGEMAP

```
1: function EDGEMAP(U, H, F, M, C, R)
2:     if (the density of $U >$ threshold) then
3:         return EDGEMAPDENSE (U, H, F, M, C)
4:     else
5:         return EDGEMAPSPARSE(U, H, F, M, C, R)
```

Algorithm 5 EDGEMAPDENSE

```
1: function EDGEMAPDENSE(U, H, F, M, C)
2:     $Out = \{\}$
3:     parfor $v.id \in V$ do
4:         $v^{new} = v$
5:         for $(ngh, v) \in H$ do
6:             if $(C(v^{new}) == 0)$ then break
7:             if ($ngh.id \in U$ and $F(ngh, v^{new}) == 1$) then
8:                 $v^{new} = M(ngh, v^{new})$
9:                 Add $v.id$ to $Out$
10:     return $Out$
```

Algorithm 6 EDGEMAPSPARSE

```
function EDGEMAPSPARSE(U, H, F, M, C, R)
    $Out = \{\}, Tmp = \{\}$
    parfor $u.id \in U$ do
        parfor $(u, ngh) \in H$ do
            if $(C(ngh) == 1$ and $F(u, ngh) == 1)$ then
                Add $(M(u, ngh), ngh.id)$ to $Tmp$
                Add $ngh.id$ to $Out$
    parfor $v.id \in Out$ do
        $v^{new} = v$
        for $(t, v.id) \in Tmp$ do
            $v^{new} = R(t, v^{new})$
    return $Out$
```

IV. System Design and Implementation

To realize the programming model above, we design and implement a new distributed framework. In this section, we first describe the system architecture. The technique details of the main components will also be presented. And then, we will introduce some system optimizations we have explored.

There are several main components in the framework, as shown in Figure 2. The first is a code generator which takes the high-level FLASH APIs as input, and generates execution code to be run on the second component named FLASHWARE, which is a middleware designed and optimized for the distributed graph processing. The FLASHWARE executes the code produced by the code generator on the distributed runtime, which contains multiple CPUs of the cluster machines. Each process acts as an individual worker and could contain multiple working threads. Each worker processes a fragment of the distributed computation, and the communication between different workers is implemented through MPI.

A. Graph Partition

The graph is partitioned using an *edge-cut* scheme, as we described in Section II. Every worker is assigned with a set of disjoint vertices (and the associated edges). For a worker, the vertex data is held in its memory. While for the edges, it depends: if there are enough memory capacity, the edges are cached in memory all the time; otherwise, they are only loaded from disks when necessary. The built-in partitioner is hash-based as it is most-frequently used, and it is more effective than other lightweight strategies in FLASH according to our evaluation (in the full version [1]). FLASH also allows the users for specifying their own edge-cut partition strategy (e.g., METIS [36]) to pursuit even better workload balance. As this is not the focus of this work, we do not dive into more details. R2.O1, R3.O3

Suppose an m-worker cluster, the given graph $G = (V, E)$ will be partitioned into m partitions. V_i is the vertex subset held by the i-th worker, with each vertex in V_i owns a *master* on this worker. There are also mirrors created for remote vertices (i.e., vertices assigned to other workers). Figure 2 gives an example, where the first three vertices locate at worker #1, and the other three vertices belong to worker #2. For each *vertexSubset*, a worker simply maintains a set of vertex *ids*, representing the master vertices in the set that locate on it.

B. FLASHWARE

FLASHWARE is designed as a middle layer that completes intra-node updating and inter-node communication (message passing). FLASHWARE enables FLASH's interface to hide the details of communication and data distribution, as well as provides the capability to apply multiple system optimizations automatically and adaptively at the runtime.

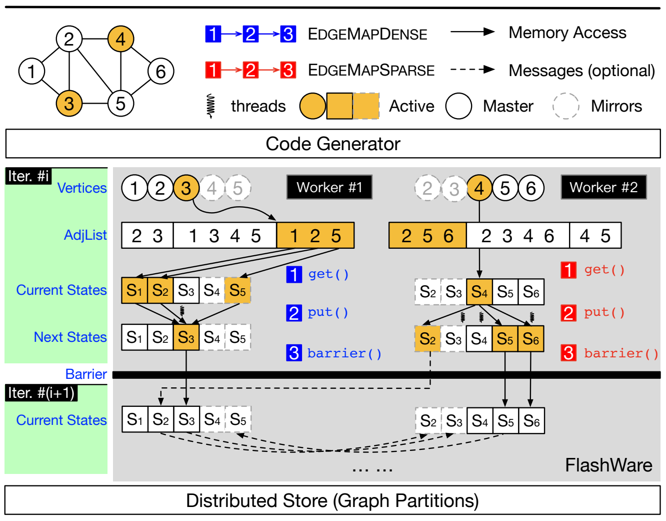


Fig. 2: System architecture and the design of FLASHWARE.

Data access. FLASH's execution is based on the BSP model, hence FLASHWARE distinguishes the *current states* and the *next states*. The current states of a vertex are ensured to be consistent on all workers who access it in the current superstep. On the other hand, the updated values are written in the next states, which is not visible to other vertices in the current superstep and may be different between workers. To save the memory, the next states are only created when necessary.

FLASHWARE exposes two APIs for accessing data:

(1). $\textbf{\textit{get}}(id : \mathbb{N}) \mapsto vertex$

(2). $\textbf{\textit{put}}(id : \mathbb{N},$
$v : vertex,$
$R(new : vertex, old : vertex) \mapsto vertex) \mapsto None$

The *get* function is used to access a specific vertex with *id* through reading the current states. And the *put* function is used to update the vertex with *id* using a new value v through writing the next states. The data in current states is read-only and consistent during a superstep, thus arbitrary vertices (masters or mirrors) can be obtained safely without data races. As for *put*, the operations on a master are applied immediately, while operations on a mirror may incur concurrent updates (in EDGEMAPSPARSE), so a reduce function R is required to aggregate the new value with the old value. This kind of modification is first applied on the mirror inside this worker, and then further propagated to the corresponding master. M1, R2.O1, R3.O1

Communication/Synchronization. Both of VERTEXMAP and EDGEMAPDENSE make updates for the masters, thus messages are only from masters to related mirrors. While for EDGEMAPSPARSE, the synchronization procedure is three-phased: a mirror first merges updates to form a temporary new value for this vertex; then mirrors send messages to the master which processes these messages to decide the final state; and a master broadcasts its final state to necessary mirrors. As a result, there are two rounds of message-passing.

(3). *barrier*() $\mapsto$ $None$

The interface $barrier()$ acts similar to $MPI\_Barrier$. It is called at the end of a superstep, forcing the workers to wait until all workers have completed the processing for the current superstep. It ensures that all messages are delivered and all updates are applied. When the *barrier* function finishes, the current states and the next states are swapped to make all updates in this superstep to be visible in the current states for starting a new superstep.

R2.O1, R3.O3 **Fault Tolerance.** Inspired by GraphLab [8], it is possible to achieve fault tolerance for FLASH by constructing the snapshots of FLASHWARE, because the system runs based on the states of FLASHWARE. For constructing a snapshot, the modified data of vertex states since the last snapshot as well as all *vertexSubsets* are saved. Once there is a failure, the system will be recovered from the last checkpoint. We are currently implementing this distributed checkpoint mechanism as an important future work for the production deployment.

C. Code Generation

The code generator generates code to be executed by FLASHWARE, from the high-level user-provided APIs. It also enables some optimizations through static analysis of the program, e.g., to decide the critical attributes, as we will discuss in the next section. Take the VERTEXMAP as an illustration, we now explain the logics of the code generation. A VERTEXMAP contains calling $get()$ and $put()$ on the master vertices in parallel inside a worker. Once a master is updated, messages are generated adaptively to synchronize the new state to its mirrors. The $barrier()$ is called at last to ensure that local computation is completed on each worker, and all messages are delivered. After that, the superstep for this VERTEXMAP finishes. The code generation for EDGEMAPDENSE is similar, except that it could call get on mirror vertices, instead of the master vertices only, as Figure 2 shows. the EDGEMAPSPARSE is more complex since it also calls put on mirror vertices.

D. Optimizations

M2, R1.O2, R2.O2, R3.O1 **Overlap communication with computation.** For a worker with c cores, it maintains a thread pool containing c threads. Among them, one thread is responsible for inter-worker message sending and one is for receiving via MPI. The other threads perform parallel vertex-centric processing. As separate threads are created to execute message passing, the communication is overlapped with computation, leading to the performance improved. Take the VERTEXMAP function as an example, once a vertex has been processed and updated (i.e., applying the function M on its master), the new value will be immediately added to the buffer which contains messages to be sent. The sending thread continually picks up messages from the sending buffer, and send the batched messages to another worker. At the same time, the receiving thread checks if there are messages from each of the other workers in turn. In this way, the computation and communication tasks are co-scheduled. This technique can speed up all kinds of algorithms, and the effect depends on the percentages of communication and computation in total time. An experimental analysis is given in Section V-D.

TABLE II: The rules to decide critical properties.

	VERTEXMAP	EDGEMAPDENSE		EDGEMAPSPARSE	
		source	target	source	target
get	×	✓	×	×	✓
put	×	–	×	–	✓

"✓" means that the property is decided to be critical; "×" means that cannot decide yet; "–" means that this kind of operation never happens.

Synchronize critical properties only. In some cases, there are multiple vertex properties, but not all of them are *critical*. We say a property critical only if it is accessed by other vertices, thus the update to the master need to be broadcasted to its mirrors. On the contrary, if a property is only read by the master, means that it is only useful in local computation, it is not critical. Therefore, the mirrors do not hold such uncritical properties. Given a vertex property, we can decide if it is critical through static analysis of the program before the execution. Table II gives the rules based on a classification of the operations on a property. If and only if it is got as the property of the source vertex in an EDGEMAPDENSE function, or it is got/putted as the property of the target in an EDGEMAPSPARSE function, it is critical.

This optimization reduces the communication cost: (1) when there are critical properties being updated, it makes a reduction on a single message's size from the total size of all properties to only that of critical properties; (2) when only uncritical properties are updated, no need of synchronization for them. In GC and LPA, the set on each vertex for recording neighbors' colors is an uncritical property; in TC and RC, each vertex maintains an uncritical property representing the count which it is involved; in MIS, the property representing if it is available ($v.b$ in Algorithm 13 in the full version) is uncritical. For other applications (e.g., CC, BFS and BC) that do not contain uncritical properties, this technique has no benefits.

Communicate with necessary mirrors only. Another intuitive way to eliminate redundant messages is to communicate with only the necessary mirrors. For normal graph applications, the messages are transferred along the edges. Therefore, the master of a vertex should only communicate with its mirrors that located in a partition which contains at least one neighbor of this vertex. For synchronizing a single vertex, it reduces the number of messages from m to m', where m is the number of partitions and m' is the number of necessary mirrors. In fact, the average value of m' is the replication factor which is determined by the partition strategy. According to [37], when partitioning the TW graph into 48 partitions using a hash-based strategy, the replication factor is 16.0, means that two-thirds of messages can be reduced. Only in the cases that the programmers define virtual edges for EDGEMAP, which beyond the scope of E (e.g., in CC-log), FLASHWARE synchronizes the update on a master to the mirrors in all partitions, thus this optimization is disabled.

V. Evaluation

In this section, we present the evaluation results to show the superiority of FLASH over previous works in terms of expressiveness, productivity, and efficiency. (1) For expressiveness, we choose 18 representative graph applications from [20] and implement them in FLASH. Each of them represents a general category of algorithms and is frequently used in real-world applications. Some applications are rarely provided in previous works, indicating FLASH's strong expressiveness. (2) For productivity, we try our best to implement algorithms for these applications in state-of-the-art graph frameworks, and show the *Logical Line of Codes* (LLoCs) [27]. Due to the limitations of the expressiveness in other works, we can only implement the sub-optimal version or even failed to implement the naivest version for some algorithms. (3) For efficiency, we compare the execution time of FLASH with other frameworks, which shows that FLASH achieves the best overall performance. We also conduct micro benchmarks to reveal why FLASH runs faster in most of cases.

A. Experimental Setup

We compare FLASH with state-of-the-art graph processing frameworks using following setups:

Platform. All our experiments are conducted on a 20-node Intel(R) Xeon(R) CPU E5-2682 v4 based system. Each node has 32 cores running at 2.50 GHz and 512GB of main memory. All nodes are connected with a 10Gb ethernet. In order to perform the comparison, the latest version of Pregel+, PowerGraph, Gemini and Ligra are installed on the cluster. We leverage OpenMP to manage threads and Open MPI to manage message passing between nodes.

Datasets. We use a collection of real-world graphs, including social networks (SN), web graphs (WG), and road networks (RN), with their basic characteristics illustrated in Table III. For the social network, it is characterized by a very skew distribution of edges, usually with some "hot" vertices having an extremely high degree. For the road network, it has a very long diameter and fewer associated edges per vertex. And the web graph lies somewhere in middle. For unweighted graphs, random weights are added to each edge if necessary.

Applications. We choose 18 representative graph applications to demonstrate FLASH's ability for programming distributed graph algorithms effortlessly and effectively, as shown in Table IV. CC and BFS are well supported by all tested frameworks, because the ISVP algorithms perform well enough for most cases. We also implemented an optimized CC algorithm [19] in FLASH since it performs better on large-diameter graphs. For BC, MIS and MM, every system we evaluated is able to express a basic algorithm correctly, but they failed to express the advanced versions, suffering either poor performance or complicated programs. FLASH is not only able to implement the basic version with less effort, but also make the advanced version possible. For KC, TC and GC, even the naivest algorithms are not feasible to implement in some frameworks, while FLASH is able to implement them with less effort, because of the support for non-ISVP algorithms. As for SCC, BCC, LPA, MSF, they are rarely provided by existing graph frameworks, and it is often failed to implement them. While it is easy for FLASH to do so. The last six applications result in local algorithms, in which a piece of message will be propagated within only several hops. Since this kind of workloads are not the main target of general graph processing frameworks, few of them are provided by the competitors, which verified the expressiveness of FLASH once again. M3, R1.O4

TABLE III: A collection of real-world graphs.

Abbr.	Dataset	|V|	|E|	Diameter	Domain
OR [38]	soc-orkut	3.07M	117M	9	SN
TW [39]	soc-twitter	41.7M	1.47B	15	SN
US [40]	road-USA	23.9M	28.9M	1452	RN
EU [40]	europe-osm	50.9M	54.1M	2037	RN
UK [40]	uk-2002	18.5M	298M	25	WG
SK [40]	sk-2005	50.6M	1.95B	23	WG

TABLE IV: A collection of representative graph applications.

Abbr.	Application	Abbr.	Application
CC	connected components	BFS	breadth-first search
BC	betweenness centrality	MIS	maximal independent set
MM	maximal matching	KC	k-core decomposition
TC	triangle counting	GC	graph coloring
SCC	strongly CC	BCC	biconnected components
LPA	label propagation	MSF	minimum spanning forest
RC	rectangle counting	CL	k-clique counting
3PC	3-path counting	DC	diamond counting
TTC	tailed-triangle counting	KCS	k-core searching

Baselines. Four representative state-of-the-art graph processing frameworks are tested as the baselines: Pregel+, PowerGraph, Gemini and Ligra. Pregel+ [13] is the representative framework that uses Pregel's vertex-centric model. Compared with other Pregel-like frameworks (e.g., Giraph [7] and GPS [12]), Pregel+ provides simpler interface and higher efficiency. PowerGraph [9] is the representative framework that adopts the GAS model. Gemini [11] is a state-of-the-art framework, which is reported to significantly outperform all well-known existing distributed graph processing frameworks, however, its expressiveness is weaker than other vertex-centric frameworks. Ligra [21] is a shared-memory framework which provides similar interfaces with FLASH.

These frameworks provide some pre-optimized built-in algorithms, but none of them provides implementations for all applications in Table IV, due to their limitations of expressiveness. For fair comparison, we test the built-in algorithms if provided, or, we try our best to implement them in these frameworks. Note that some frameworks provide multiple implementations for an application, and we report the best performance of these variants when conducting tests on them.

B. Overall Performance

Table V reports the results on overall execution time of the first eight applications on six datasets. All these tests are conducted on 4 nodes of the cluster except Ligra, which only uses a single node. For fair comparison, the initial pre-

TABLE V: Execution time for the first eight applications on six datasets (in seconds).

App.	Data	Pregel+	PowerG.	Gemini	Ligra	FLASH	App.	Data	Pregel+	PowerG.	Gemini	Ligra	FLASH
CC	OR	9.21	5.31	1.24	0.49	**0.48**	BFS	OR	3.07	6.27	0.87	**0.35**	**0.35**
	TW	99.31	281.93	8.60	10.09	**6.38**		TW	31.47	48.11	4.61	**2.28**	6.16
	US	435.42	1832.2	524.34	323.43	**30.96**		US	202.79	1512.3	519.01	244.01	**12.17**
	EU	1740.0	6749.7	1302.3	663.10	**76.47**		EU	1035.5	4453.4	1445.4	506.72	**50.32**
	UK	33.56	26.33	3.33	**2.09**	2.51		UK	5.94	15.51	2.78	**1.09**	2.26
	SK	132.97	307.30	5.57	**4.07**	7.02		SK	29.33	35.96	3.53	**1.92**	6.02
BC	OR	11.23	13.40	1.73	0.81	**0.54**	MIS	OR	11.22	12.30	1.78	2.66	**0.51**
	TW	110.29	121.71	**8.15**	21.62	11.77		TW	55.62	176.77	4.66	20.61	**4.58**
	US	516.86	3066.8	1007.1	411.25	**16.94**		US	4.55	22.58	3.93	1.10	**0.94**
	EU	2981.1	OT	2861.8	978.21	**129.64**		EU	254.88	722.41	188.22	122.41	**12.14**
	UK	22.61	39.91	6.24	**2.18**	3.87		UK	14.05	65.64	20.46	4.92	**1.83**
	SK	116.13	127.23	7.54	**7.08**	11.49		SK	77.54	108.54	13.37	9.24	**5.13**
MM	OR	OT	OT	497.15	889.61	**22.27**	KC	OR	678.44	1140.6	–	302.65	**4.03**
	TW	OT	OT	OT	OT	**25.15**		TW	4937.4	OT	–	1313.4	**29.26**
	US	13.00	65.66	6.96	3.69	**3.03**		US	232.18	68.80	–	16.11	**2.12**
	EU	428.87	1547.7	253.25	182.36	**19.17**		EU	OT	634.68	–	195.04	**10.44**
	UK	OT	OT	1091.8	518.83	**22.11**		UK	2924.6	2682.4	–	577.72	**5.38**
	SK	OT	OT	OT	OT	**114.76**		SK	OT	OT	–	3702.8	**44.16**
TC	OR	529.61	27.86	–	12.90	**3.32**	GC	OR	OT	13.26	–	–	**9.72**
	TW	OOM	720.01	–	OT	**49.10**		TW	OT	426.37	–	–	**264.44**
	US	17.90	6.48	–	**0.57**	1.09		US	10.29	13.11	–	–	**2.38**
	EU	32.56	10.91	–	**0.53**	2.29		EU	242.59	**43.81**	–	–	54.61
	UK	OOM	17.44	–	14.23	**7.00**		UK	2219.7	36.19	–	–	**35.67**
	SK	OOM	211.67	–	OT	**70.59**		SK	OT	706.21	–	–	**331.72**

"–" means that we fail to implement an available algorithm for this case because of the limitations in expressiveness; "OOM" means that the tested algorithm failed due to exhausted memory. "OT" means that the execution did not terminate within 5000s.

TABLE VI: Execution time for four more complex applications on six datasets (in seconds).

App.	Data	Baseline	FLASH	App.	Data	Baseline	FLASH
SCC	OR	120.76	**1.24**	BCC	OR	303.93	**5.57**
	TW	949.60	**13.80**		TW	3615.0	**75.85**
	US	719.91	**57.84**		US	3844.7	**169.58**
	EU	3021.1	**161.35**		EU	OT	**486.14**
	UK	223.22	**5.55**		UK	879.91	**22.82**
	SK	1335.5	**18.26**		SK	2991.8	**55.20**
LPA	OR	155.90	**16.83**	MSF	OR	55.96	**6.96**
	TW	1433.9	**100.31**		TW	867.54	**72.51**
	US	49.11	**2.77**		US	**25.42**	29.96
	EU	276.20	**25.57**		EU	**64.86**	68.66
	UK	299.62	**11.06**		UK	55.25	**29.74**
	SK	OT	**78.25**		SK	477.72	**86.84**

The baseline results for SCC, BCC and MSF are tested on Pregel+, and the baseline results for LPA are tested on PowerGraph.

TABLE VII: Execution time of FLASH for six local applications on six datasets (in seconds).

Data	RC	CL	3PC	DC	TTC	KCS
OR	12.49	20.33	12.51	31.50	12.12	2.03
TW	140.16	OT	OOM	OOM	OOM	7.45
US	1.31	1.22	1.29	1.93	1.40	2.22
EU	2.75	2.39	2.46	3.24	2.48	2.20
UK	14.65	420.12	27.09	77.23	27.20	2.21
SK	176.78	OT	OOM	OOM	OOM	18.17

For KCS, $k = 3$ on US and EU and $k = 50$ on others. As for CL, $k = 4$.

processing time (for data loading and partitioning) and post-processing time (e.g., writing the results) of every framework are not recorded. Specially, Pregel+ may decompose the algorithm (*e.g.,* BC, SCC, and BCC) into several individual sub-algorithms and chain them by taking the output of the previous as the input of the next. In this situation, the data sharing time among sub-algorithms will be recorded. Table VI reports the results of the four more complicated applications. Since they are difficult or not supported to be implemented in previous works, we only compare the performance of FLASH with the most efficient implementation provided by other frameworks.

In 95.2% cases, FLASH provides competitive performance compared with the one that performs best (within $2\times$ slow-down); and in 84.5% cases, it is faster than all other compared frameworks, with the speedup up to 2 orders of magnitude. For example, when calculating MM on the TW dataset, FLASH only takes 25.15 seconds, while all the other frameworks failed to get the results within 5000 seconds. Another example is SCC, which requires complex computation and is only provided by Pregel+ as far as we know. The implementation in Pregel+ is $22.7\times$ to $54.6\times$ slower than FLASH. PowerGraph performs efficiently on GC since it implements an asynchronous algorithm, which converges faster than a BSP-based algorithm. Ligra is faster than FLASH in some cases because it is a shared-memory system, with the communication cost much cheaper than that of distributed systems.

To further validate FLASH's expressiveness, some local algorithms are evaluated. Besides TC, which is provided by some compared frameworks, we implemented algorithms in FLASH for matching other subgraphs including rectangles, 3-paths, tailed triangles and diamonds. Searching for k-cliques and k-cores are also implemented in FLASH without much effort. Table VII reports the time performance. Since this kind M3, R1.O4

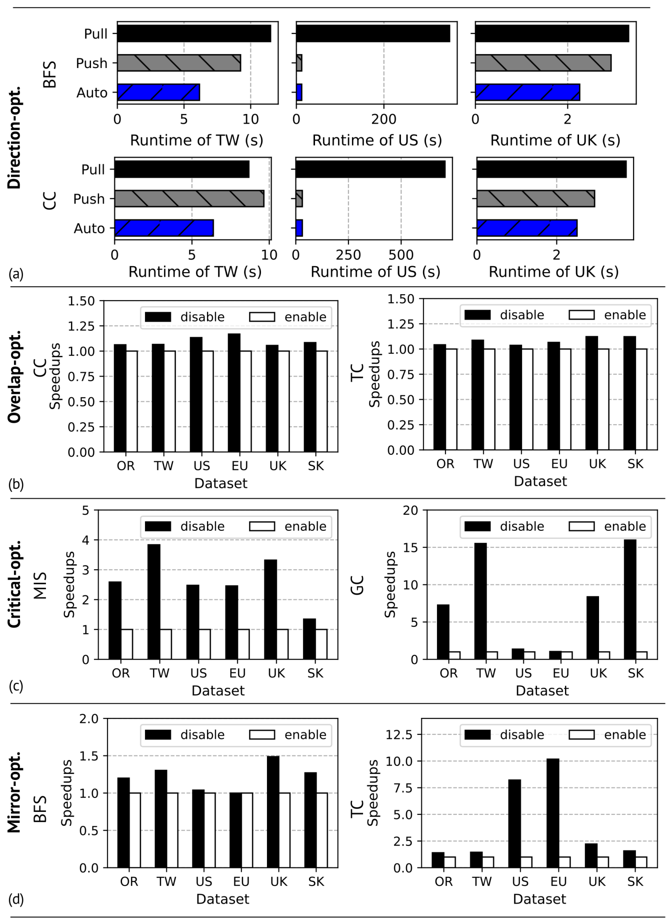


Fig. 3: The effectiveness of (a) dual-direction, (b) overlap, (c) critical properties and (d) necessary mirrors optimization.

of local algorithms are often non-ISVP, other general graph processing frameworks mainly focus on global algorithms (e.g., BFS, CC and PageRank), while FLASH is suitable for both global and local algorithms.

C. Productivity

To demonstrate the productivity of FLASH, the LLoCs are counted, as shown in Table I. Note that we only consider LLoCs in the core functions, while ignoring the comments, input/output expressions, and data structure (e.g., the graph) definitions. Gemini fails to express most algorithms since its programming model is most limited. PowerGraph needs lots of code for TC since it does not provide the serialization/de-serialization semantics for users to exchange neighbor-lists. Although Pregel+ is able to express some complex non-ISVP algorithms (e.g., SCC and BCC), it is usually intractable. In fact, its algorithms for these applications are decomposed into several parts, with each part constitutes of an individual sub-algorithm and needs all necessary functions to be programmed. This is obviously not friendly to the users. Moreover, since every sub-algorithm needs its own implementation for parsing the input from the previous sub-algorithm and outputting data for the next sub-algorithm (which are not counted in the LLoCs), the actual lines of code are further more than the results in Table I (811 lines in total for SCC and 3017 lines for BCC). This algorithm decomposition also results in poor performance.

Although all compared frameworks evaluated are claimed to provide succinct interface, and have been widely used, FLASH requires less effort in implementation than other works in all tested cases, showing that FLASH achieves the best productivity when expressing both the ISVP and non-ISVP algorithms. We provide more examples in the full version [1] for the readers to further judge the succinctness and readability.

D. Micro Benchmarks

As analyzed in Section IV-D, not all optimization techniques can speed up all graph algorithms, the effects depend on the characteristics of the algorithms and the datasets. To further evaluate how these techniques impact the performance of FLASH, we conduct micro benchmarks for each of them. M2, R1.O2, R2.O2, R3.O1

Dual update propagation model. Adaptive switching between the *pull* (dense) mode and the *push* (sparse) mode according to the density improves the performance of FLASH significantly. For algorithms such as BFS and BC, in common cases, the active set is initially sparse, switches to dense after a few iterations and then switches back to sparse later. For another category of algorithms (e.g., CC, MM, MIS and GC), the active set starts as dense, and becomes sparser as the algorithm continues. For TC and PageRank, all vertices are active in each step (i.e., in the dense mode all the time), so this adaptive switching is disabled. We demonstrate the effectiveness of adaptive switching by compare its performance with using either *push* or *pull*. Figure 3 (a) shows the execution time of BFS and CC on three graphs, as expected, the performance gap is quite significant. For TW and UK, our dual update propagation scheme always achieves the best performance. As for the US graph, since it is sparse with a very low average degree, our adaptive switching falls into the sparse mode all the time, while the dense mode consumes much longer execution time.

Overlap communication with computation. We compare the execution time of different algorithms with and without this optimization, as shown in Figure 3 (b). According to our evaluation, for algorithms that incur a heavy computation overhead (e.g., TC and CL), this optimization leads to a performance improvement of $4\% \sim 12\%$. And for algorithms that the computation is lightweight and the communication time accounts to more percentages, e.g., CC and MM, it leads to a performance improvement of $6\% \sim 23\%$.

Synchronize critical properties only. As analyzed in Section IV-D, this strategy is useful when there are uncritical properties. Figure 3 (c) shows the execution time of MIS and GC on different graphs when enabling and disabling this optimization, showing that it could lead to a considerable speedup. These two algorithms represent different workloads. In MIS, the uncritical property is a 1-byte boolean variable, while in GC, the uncritical property is the neighbor set which can be very large. Therefore, the performance improvement on GC is more significant, up to $16.0\times$.

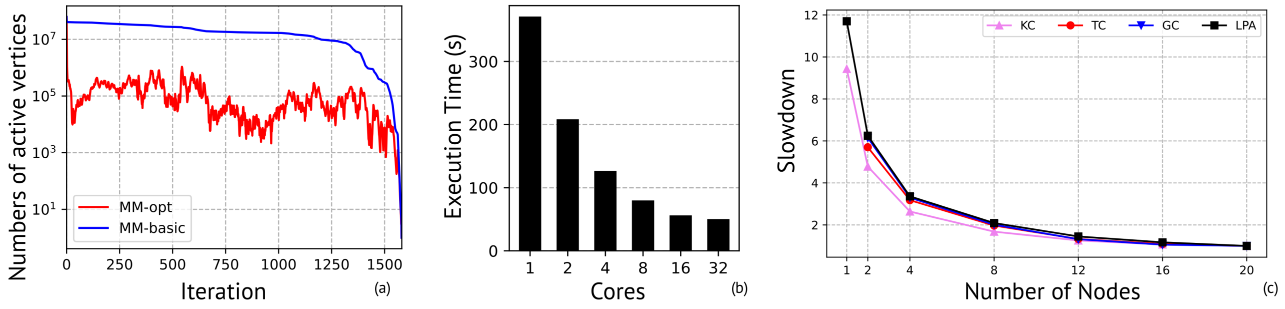


Fig. 4: (a) The number of active vertices for MM-basic and MM-opt on TW. (b) Performance of TC on TW with varying cores per node (4 nodes in total). (c) Performance of KC, TC, GC and LPA on TW with varying nodes (4 cores per node).

Communicate with necessary mirrors only. To evaluate the effect of communicating with only necessary mirrors, we report the execution time of BFS and TC when enabling and disabling this optimization (Figure 3 (d)). BFS represents algorithms that contain small messages, while TC represents other ones in which the messages can be very large (the neighbor set). With a reduction on the number of messages, this technique leads to a speedup of $1.0\times \sim 1.5\times$ for BFS and $1.4\times \sim 10.2\times$ for TC.

E. Advanced Implementations

The high expressiveness of FLASH does not only allow users to implement algorithms with less effort, but also allows users to implement the advanced version of some algorithms for higher performance. Consider the MM application as an example, we implement a basic algorithm in FLASH (MM-basic), as well as an optimized one (MM-opt), as described in the full version. Other frameworks cannot implement MM-opt since they do not support to define arbitrary edges sets. The advanced algorithm has higher efficiency than the basic one, which means it will touch less vertices and edges during the execution. Figure 4 (a) compares the number of active vertices (size of the frontier) in all iterations for both algorithms on the TW dataset. R3.O5 The significant reduction in active vertices leads to a considerable speedup of $70.1\times$ (1763.0s for MM-basic and 25.15s for MM-opt). This is also the main reason that FLASH significantly outperforms other works. Another example is the application CC, for which the CC-basic algorithm takes 169.6s on US, and the CC-opt only takes 30.96s. For applications that FLASH implements the same algorithm as other systems such as BFS and TC, it provides a comparative performance, as shown by Table V.

F. Scalability

Now, we examine the scalability of FLASH in terms of both intra-node and inter-node. We first compare the performance of FLASH while varying the cores of each node as 1, 2, 4, 8, 16, 32. Figure 4 (b) presents the execution time of running TC on the TW dataset using 4 nodes, which shows a reasonable trend of scalability, achieving speedup of $1.8\times$, $2.9\times$, $4.7\times$, $6.7\times$ and $7.5\times$ at 2, 4, 8, 16, 32 cores, respectively. The other cases show similar trends, except the cases that the communication time dominates the execution such as running BFS on the US graph, on which the scalability is poor for all frameworks. The reduction on execution time after 4 cores slows down since when more cores are used, the scheduling cost and memory contention inside a node increase.

M4, R2.O3, R3.O3 We also conduct experiments to evaluate the inter-node scalability using up to 20 nodes, with results shown in Figure 4 (c). All execution time are normalized to slowdown of the best case in question. When increasing the cluster size from 1 node to 20 nodes, the speedup is $11.7\times$ for LPA and $9.4\times$ for KC (TC and GC failed with 1 node). The reduction on execution time becomes slower with the number of nodes increases, because the computation time reduces but the communication cost may increase. This is also the common pattern that limits the scalability of all kinds of distributed frameworks. A piecewise breakdown analysis on the execution time validates this, showing that with the increase of the cluster size, the computation time decreases nearly linearly, while the communication time (for message-passing and serialization) accounts to more and more percentages of the total time. For example, the computation time for TC with 4 nodes is $3.6\times$ of that with 16 nodes, while the communication time is $1.2\times$.

VI. Conclusions and Future Work

In this paper, we present FLASH, a framework for programming distributed graph algorithms. We track three essential metrics for graph frameworks from exploring more advanced and complex graph algorithms: expressiveness, productivity and efficiency. Providing a new high-level programming model and an efficient system implementation, FLASH leads to easy programming for a wide variety of graph algorithms, and achieves outstanding performance at the same time. In addition to the algorithms discussed in this paper, we believe that other algorithms can also benefit from our framework, since the huge potentials it revealed.

R3.O6 As a topic for future work, FLASH is proposed to be a component of GraphScope (the unified framework for large-scale graph processing at Alibaba) for easing the programming. Meanwhile, GRAPE [16] is the current component for executing graph algorithms in GraphScope. While GRAPE focuses on the low-level execution of graph algorithms, FLASH emphasizes their easy programming in the higher level. Their technique stacks are thus orthogonal in that a FLASH program can be compiled into the GRAPE runtime to combine their advantages, which will also be our future proposal.

REFERENCES

[1] (2022) Flash: A framework for programming distributed graph processing algorithms. [Online]. Available: https://graphscope.io/flash-full.pdf

[2] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[3] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," *Proc. VLDB Endow.*, vol. 6, no. 4, p. 265–276, Feb. 2013.

[4] J. Domke, T. Hoefler, and W. E. Nagel, "Deadlock-free oblivious routing for arbitrary topologies," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 616–627.

[5] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 437–448.

[6] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD International Conference on Management of Data*, 2010, pp. 135–146.

[7] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," in *Proceedings of the Hadoop Summit*, vol. 11, no. 3, 2011, pp. 5–9.

[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *arXiv preprint arXiv:1204.6078*, 2012.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, 2014, pp. 599–613.

[11] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 301–316.

[12] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th international conference on scientific and statistical database management*, 2013, pp. 1–12.

[13] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1307–1317.

[14] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan, "Computation and communication efficient graph processing with distributed immutable view," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 215–226.

[15] T. Li, C. Ma, J. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, and I. Raicu, "Graph/z: A key-value store based scalable graph processing system," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 516–517.

[16] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu, "Parallelizing sequential graph computations," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 4, pp. 1–39, 2018.

[17] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 2, pp. 305–324, 2017.

[18] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: graph algorithms for the (semantic) web," in *International Semantic Web Conference*. Springer, 2010, pp. 764–780.

[19] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in mapreduce," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 827–838.

[20] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[21] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[22] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[23] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[24] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[25] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "Effective and efficient dynamic graph coloring," *Proceedings of the VLDB Endowment*, vol. 11, no. 3, pp. 338–351, 2017.

[26] J. Shi, L. Dhulipala, and J. Shun, "Parallel clique counting and peeling algorithms," 2020. [Online]. Available: https://arxiv.org/abs/2002.10047

[27] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A sloc counting standard," in *Cocomo ii forum*, vol. 2007. Citeseer, 2007, pp. 1–16.

[28] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, aug 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[29] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, 2014.

[30] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–53, 2018.

[31] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, 2010, pp. 303–314.

[32] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 68–70.

[33] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, pp. 35–41, 1977.

[34] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 456–471.

[35] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2015, pp. 183–193.

[36] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," 01 1995.

[37] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[38] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[39] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.

[40] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[41] M. Luby, "A simple parallel algorithm for the maximal independent set problem," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC '85. New York, NY, USA: Association for Computing Machinery, 1985, p. 1–10. [Online]. Available: https://doi.org/10.1145/22145.22146

[42] T. Schank, "Algorithmic aspects of triangle-based network analysis," *Phd in computer science, University Karlsruhe 3*, 2007.

[43] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, "k-core organization of complex networks," *Phys.rev.lett*, vol. 96, no. 4, pp. 185–194, 2006.

[44] H. N. Gabow and H. H. Westermann, "Forests, frames, and games: algorithms for matroid sums and applications," in *STOC '88*, 1988.

[45] E. C. Freuder, "A sufficient condition for backtrack-free search," *Journal of the ACM*, vol. 29, no. 1, pp. 24–32, 1982.

[46] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the Vldb Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[47] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, “Pregel algorithms for graph connectivity problems with performance guarantees,” *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1821–1832, 2014.

[48] S. Orzan, “On distributed verification and verified distribution,” Ph.D. dissertation, Vrije Universiteit Amsterdam, 2004, naam instelling promotie: Vrije Universiteit.

[49] G. M. Slota, S. Rajamanickam, and K. Madduri, “Bfs and coloring-based parallel algorithms for strongly connected components and related problems,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 550–559.

[50] X. Zhu and Z. Ghahramani, “Learning from labeled and unlabeled data with label propagation,” Tech. Rep., 2002.

[51] U. N. Raghavan, R. Albert, and S. Kumara, “Near linear time algorithm to detect community structures in large-scale networks,” *Physical Review E*, vol. 76, no. 3, sep 2007. [Online]. Available: https://doi.org/10.1103%2Fphysreve.76.036106

[52] N. Christofides, “Worst-case analysis of a new heuristic for the travelling salesman problem,” Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1976.

[53] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, “The complexity of multiterminal cuts,” *SIAM Journal on Computing*, vol. 23, no. 4, pp. 864–894, 1994.

[54] K. J. Supowit, D. A. Plaisted, and E. M. Reingold, “Heuristics for weighted perfect matching,” in *Proceedings of the twelfth annual ACM symposium on Theory of computing*, 1980, pp. 398–419.

[55] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[56] S. Chung and A. Condon, “Parallel implementation of bouvka’s minimum spanning tree algorithm,” in *Proceedings of International Conference on Parallel Processing*. IEEE, 1996, pp. 302–308.

Appendix A
Expressing other programming models

FLASH has the ability to simulate the traditional vertex-centric programming models. Since other vertex-centric programming models such as GAS [9] and Gemini [11] are more limited in expressiveness compared to the original model proposed by Pregel [6], we try to simulate the Pregel-like model using FLASH. As a consequence, it is possible to port any existing vertex-centric programs in our model.

The typical execution workflow of the vertex-centric model proceeds in synchronized iterations (or *supersteps*). As shown in Algorithm 7, in each superstep, all active vertices (called *frontiers*) execute the same user-defined vertex function in parallel, which receives a set of messages as input ($inbox$) and can produce one or more messages as output ($outbox$). At the end of a superstep, the runtime receives the messages from the $outbox$ of each vertex and computes the set of active vertices for the next superstep. The execution will terminate when there are no active vertices or when some convergence condition is met.

As shown in Algorithm 8, the local computation in each superstep of the vertex-centric model can be implemented in FLASH through a VERTEXMAP function, which processes the message $inbox$ and produces the updated value as well as the message $outbox$ for each vertex, and the EDGEMAP function adds the message to the $inbox$ of the target from the $outbox$ of the source.

Algorithm 7 VERTEX-CENTRIC MODEL SEMANTICS

```
1: $A = V$
2: while SIZE($A$) $> 0$ do
3:     parfor $v \in A$ do
4:         $v.inbox$ = RECEIVEMESSAGES($v$)
5:         ($value^{new}, v.outbox$) = COMPUTE($v.value, v.inbox$)
6:         $v.value = value^{new}$
```

Algorithm 8 SIMULATING VERTEX-CENTRIC USING FLASH

```
1: function LOCAL($v$)
2:     ($v.value, v.outbox$) = COMPUTE($v.value, v.inbox$)
3:     return $v$
4:
5: function UPDATE($s, d$)
6:     add the messages (from $s$ to $d$) to $d.inbox$ from $s.outbox$
7:     return $d$
8:
9: function MERGE($t, d$)
10:    $d.inbox = d.inbox \cup t.inbox$
11:    return $d$
12:
13: $A = V$
14: while SIZE($A$) $> 0$ do
15:     $A$ = EDGEMAP($A, E$, CTRUE, UPDATE, CTRUE, MERGE)
16:     $A$ = VERTEXMAP($A$, CTRUE, LOCAL)
```

Appendix B
Graph Partition

R2.O1, R3.O3 We implement two lightweight and widely-used graph partitioning strategies: random hash-based (by hashing over vertex id) and the chunk-based [11] partitioning. The effects on performance of FLASH are evaluated by comparing the execution time of different workloads, as shown in Table VIII. In general, the hash-based strategy is more effective in FLASH since its randomness leads to more balanced workload at runtime. While in a few cases, the chunk-based partitioning reduces the overall execution time since it preserves the natural locality (e.g., running BFS on US). In fact, no single graph partition method performs best in all cases. With the default partitioner which is hash-based, FLASH enables the users to develop and apply any other effective partitioners at the cost of pre-processing overhead, such as METIS [36].

TABLE VIII: Execution time of FLASH using different partition strategies.

Data	App.	hash	chunk	Data	App.	hash	chunk
TW	BFS	6.16	**6.09**	US	BFS	12.17	**11.13**
	CC	**6.38**	6.40		CC	**30.96**	31.96
	BC	**11.77**	12.03		BC	**16.94**	17.22
	KC	**29.26**	32.44		KC	**2.12**	2.88
	MM	**25.12**	26.42		MM	**3.03**	3.18
	MIS	4.58	**4.40**		MIS	**0.94**	1.12

Appendix C
Applications List

We have implemented 72 graph algorithms in FLASH for 49 different common problems. Now we list all the algorithms in Table IX. R1.O3

Appendix D
Example Applications

Besides BFS and BC, whose algorithms are demonstrated in Section III, we now give more example applications and describe their implementations in FLASH, to show the strong expressiveness of our programming model.

A. Connected Components (CC)

A weakly connected component is a maximal subgraph of a graph such that for every pair of vertices in it, there is an undirected path connecting them. In existing vertex-centric models, the standard method for calculating CC is label propagation. In this algorithm, each vertex is attached with a property which represents its component label, being its own vertex id initially. In the subsequent supersteps, a vertex will update its label if it receives a smaller id and then it propagates this id to all its neighbors. This ISVP-based algorithm is both simple and scalable, but not necessarily efficient. As the label is propagated only one hop at a time, it may require many iterations to converge, especially for graphs that have large diameters. This algorithm is implemented in FLASH as Algorithm 9 shows.

An optimized CC algorithm [19] is proposed to overcome the problem of Algorithm 9. It utilizes a parent pointer $p(v)$ for each vertex to maintain a tree (forest) structure, with each rooted tree represents a connected component of the graph. In each iteration, the algorithm uses $StarDetection$ to identify stars (tree of depth one), in which every vertex points to one common rooted vertex (the rooted vertex is

TABLE IX: The algorithms implemented in FLASH.

Category	Applications/Algorithms [20]
Centrality	betweenness centrality (two versions), closeness centrality, eigenvector centrality, harmonic centrality, Katz centrality
Clustering and Community	clustering coefficient, ego network, fluid community, graph coloring, label propagation, label propagation by color
Connectivity	biconnected components (two versions), bridge detection (two versions), connected components (basic, optimized, block-based, two versions for cc-log, union-find based), strongly connected components (two versions)
Core	(i, j)-core (bipartite), degeneracy ordering, k-core graph decomposition, k-core searching, onion-layer ordering
Matching and Covering	maximal matching (basic, two optimized versions, bipartite), maximum matching for bipartite graphs (two versions), maximal independent set (two versions), minimal vertex cover (basic, greedy based, optimized greedy based), minimal dominating set (basic, greedy based, optimized greedy based), minimal edge cover, stable matching (bipartite)
Mining and Subgraph Matching	dimond counting, k-clique counting (two versions), matrix factorization (bipartite), rectangle counting, triangle counting, tailed triangle counting, 2-approximation for the densest subgraph problem, 3-path counting
Ranking	ArticleRank, hyperlink-induced topic search, PageRank, personalized PageRank
Standard Measurements	diameter approximation (two versions), minimum spanning forest (basic, Kruskal's algorithm based, block based), k-center
Traversal	breadth-first search, depth-first search, k-path, multi-source breadth-first search, Node2Vec walk, random walk, single source shortest path (basic, delta stepping)

Algorithm 9 CONNECTED COMPONENTS

```
function INIT($v$):
    $v.cc = v.id$
    return v
function CHECK($s, d$):
    return $s.cc < d.cc$
function UPDATE($s, d$):
    $d.cc = min(d.cc, s.cc)$
    return $d$

$U$ = VERTEXMAP($V$, CTRUE, INIT)
while SIZE($U$) $\neq 0$ do
    $U$ = EDGEMAP($U, E$, CHECK, UPDATE, CTURE, UPDATE)
```

self-pointing); then it merges stars that are connected via some edges using two $StarHooking$ operations; finally, it applies $PointerJumping$ to assign $p(v) = p(p(v))$. When the algorithm terminates, there must be isolated stars, each standing for one connected component with the rooted vertex as its id.

Algorithm 10 OPTIMIZED CONNECTED COMPONENTS

```
function INIT($v$):
    $v.p = v.id, v.f = v.id, v.s = false$
    return v

function UPDATE1($s, d$):
    $d.p = min(d.p, s.id)$,
    return $d$

function UPDATE2($s, d$):
    $d.s = true$, return $d$

function FILTER1($v$): return $(v.p == v.id)$ and $(v.s == false)$
function LOCAL1($v$):
    $v.p = INF$, return $v$

function FILTER2($v$): return $v.p == INF$
function LOCAL2($v$):
    $v.p = v.id$, return $v$

function LOCALS($v$):
    $v.s = true$, return $v$
function FS1($s, d$): return $s.p \neq d.p$
function M($s, d$):
    $d.s = false$, return $d$
function FS2($s, d$): return $(s.s == false)$ and $(d.s == true)$
function STARDETECTION($U$):
    $U$ = VERTEXMAP($U$, CTRUE, LOCALS)
    EDGEMAPDENSE($V, join(p, U)$, FS1, M, CTRUE)
    EDGEMAPSPARSE($U, join(join(U, p), p)$, CTRUE, M, CTRUE, M)
    EDGEMAPDENSE($V, join(p, U)$, FS2, M, CTRUE)

function FILTERH1($v$): return $v.s == true$
function LOCALH1($v, cond$):
    $v.f = (cond?v.p : INF)$
    return $v$
function H1($s, d$):
    $d.f = min(d.f, s.p)$
    return $d$
function FH2($s, d$):
    return $(s.p \neq s.id)$ and $(s.f \neq INF)$ and $(s.f \neq s.p)$
function H2($s, d$):
    $d.f = min(d.f, s.f)$
    return $d$
function FILTERH2($v$):
    return $(v.p == v.id)$ and $(v.f \neq INF)$ and $(v.f \neq v.p)$
function LOCALH2($v$):
    $v.p = v.f$, return $v$
function STARHOOKING($U, cond$):
    $U$ = VERTEXMAP($U$, FILTERH1, LOCALH1.bind($cond$))
    EDGEMAPDENSE($V, join(E, U)$, FS1, H1, CTRUE)
    EDGEMAPSPARSE($U, join(U, p)$, FH2, H2, CTRUE, H2)
    VERTEXMAP($U$, FILTERH2, LOCALH2)

function UPDATEJ($s, d$):
    $d.p = s.p$, return $d$
function POINTERJUMPING($U$):
    return EDGEMAPDENSE($V, join(p, U)$, CTRUE, UPDATEJ, CTRUE)

$U$ = VERTEXMAP($V$, CTRUE, INIT)
EDGEMAPDENSE($U, E$, CTRUE, UPDATE1, CTRUE)
EDGEMAPSPARSE($U, join(U, p)$, CTRUE, UPDATE2, CTRUE, UPDATE2)
$U$ = VERTEXMAP($V$, FILTER1, LOCAL1)
$U$ = EDGEMAPDENSE($V, join(E, U)$, CTRUE, UPDATE1, CTRUE)
$U$ = MINUS($V$, VERTEXMAP($V$, FILTER2, LOCAL2)
while SIZE($U$) $\neq 0$ do
    STARDETECTION($U$), STARHOOKING($A, true$)
    STARDETECTION($U$), STARHOOKING($A, false$)
    $U$ = POINTERJUMPING($U$)
```

Algorithm 11 MAXIMAL MATCHING

```
1: function INIT($v$):
2:     $v.s = -1, v.p = -1$
3:     return v
4:
5: function COND($v$): return $v.s == -1$
6: function UPDATE($s, d$):
7:     $d.p = max(d.p, s.id))$
8:     return $d$
9: function R1($t, d$):
10:     $d.p = max(d.p, t.p)$
11:     return $d$
12:
13: function CHECK($s, d$): return $(s.p == d.id)$ and $(d.p == s.id)$
14: function UPDATE2($s, d$):
15:     $d.s = s.id$
16:     return $d$
17: function R2(t, d): return $t$
18:
19: $U$ = VERTEXMAP($V$, CTRUE, INIT)
20: while SIZE($U$) $\neq 0$ do
21:     $U$ = VERTEXMAP($U$, COND, INIT)
22:     $U$ = EDGEMAP($U, E$, CTRUE, UPDATE, COND, R1)
23:     EDGEMAP($U, E$, CHECK, UPDATE2, COND, R2)
```

Algorithm 12 OPTIMIZED MAXIMAL MATCHING

```
1: function INIT($v$):
2:     $v.s = -1, v.p = -1$
3:     return v
4: function LOCAL($v$):
5:     $v.s = v.p$
6:     return v
7:
8: function F1($s, d$): return $s.s == -1$
9: function M1($s, d$):
10:     $d.p = max(d.p, s.id))$
11:     return $d$
12: function COND($v$): return $v.s == -1$
13: function R1($t, d$):
14:     $d.p = max(d.p, t.p)$
15:     return $d$
16:
17: function F2($s, d$): return $d.p == s.id$
18: function M2($s, d$):
19:     $d.s = s.id$
20:     return $d$
21: function R2($t, d$): return $t$
22:
23: function M3($s, d$): return $d$
24:
25: $U$ = VERTEXMAP($V$, CTRUE, INIT)
26: while SIZE($U$) $\neq 0$ do
27:     $U$ = VERTEXMAP($U$, COND, INIT)
28:     EDGEMAPDENSE($V, join(E, U)$, F1, M1, COND, R1)
29:     $A$ =EDGEMAPSPARSE($U, join(U, p)$, F2, M2, COND, R2)
30:     $B$ =EDGEMAPSPARSE($A, join(A, p)$, F2, M2, COND, R2)
31:     $U$ =EDGEMAPSPARSE(UNION($A, B$), $E$, F2, M3, COND, M3)
```

In this algorithm, virtual edges are generated to maintain the tree structure. Since the messages are not always along the original edges, it could not be implemented in the models that do not support communication beyond neighborhood. While in FLASH, this algorithm is expressed without much effort, as shown in Algorithm 10. Algorithm 10 converges much faster than Algorithm 9, for example, it takes only 7 iterations on the US dataset, while Algorithm 9 takes 6262 iterations. As a consequence, it brings a significant speedup up to an order of magnitude. FLASH provides some pre-defined operators for conveniently defining edge sets, for example, in this algorithm, edges with targets in U are expressed as `join(E, U)`, edges between $v \in U$ and $v.p$ are expressed as `join(U, p)` or `join(p, U)`.

B. Maximal Matching (MM)

In graph theory, a matching in an undirected graph is a set of edges without common vertices. And a maximal matching is a matching of a graph G that is not a subset of any other matching. The task of this application is to find an arbitrary maximal matching.

This problem could be solved in a greedy algorithm which always tries to build a match for each unmatched vertex (called temporary match) in the first phase. The "tie breaking" is done by following the largest vertex id. And then, if two vertices are matched each other, they will be added to the result in the second phase, means the temporary match is successful. This algorithm executes iteratively until no more vertices could be added to the result. Algorithm 11 shows how to implement this algorithm in FLASH.

Also, we can implement an optimized version for this algorithm (named MM-opt), as shown in Algorithm 12. In every step, we need to conduct computation for an unmatched vertex only if its temporary matched vertex is matched successful in the last iteration (line 28). Otherwise, its temporary matched vertex should not be changed. This is done by executing the EDGEMAPSPARSE operation from successfully matched vertices targeting specific neighbors (line 31). This algorithm is not supported by other frameworks since they do not support the users to define arbitrary edge sets.

C. Maximal Independent Set (MIS)

In graph theory, a set of vertices constitutes an independent set if and only if any two of the vertices that contained in it do not have an edge connecting them. A maximal independent set S is then a set of vertices that: (1) constitutes an independent set; and (2) there does not exist another independent set S' that is a proper superset of S (i.e., $S \subset S'$).

Maximal Independent Set (MIS) is an important and widely-used graph application, whose output is an arbitrary maximal independent set of the input graph. It is difficult to be implemented in a message-passing model and hence is not provided by most existing vertex-centric graph processing systems. To the best of our knowledge, GPS [12] is the only graph processing system that contains a distributed MIS implementation, which is based on Luby's classic parallel algorithm [41]. GPS is an open-source Pregel implementation from Stanford Infolab, which was reported to be $12\times$ faster than Giraph [7].

On the contrary, the algorithm for MIS provided by FLASH is implemented intuitively, as Algorithm 13 shows. Initially, all vertices are set to be available (to set $v.d = false$, line 22). In each step, we find all available vertices (with $v.b == true$, line 24) and add them to the result (line 25). $v.r$ is used to decide the priorities between neighbors in the same iteration. Once a vertex is added to the result, the neighbors of it should be labeled as not available immediately (to set $ngh.d = true$,

Algorithm 13 MAXIMAL INDEPENDENT SET

```
 1: function INIT($v$):
 2:     $v.d = false, v.b = true, v.r = v.deg * |V| + v.id$
 3:     return v
 4:
 5: function COND1($v$): return $v.b == true$
 6: function F1($s, d$): return $(s.d == false)$ and $(s.r < d.r)$
 7: function UPDATE($s, d$):
 8:     $d.b = false$
 9:     return $d$
10:
11: function COND2($v$): return $v.d == false$
12: function UPDATE2($s, d$): return $d$
13: function R2($t, d$):
14:     $d.d = true$
15:     return $d$
16:
17: function FILTER($v$): return $v.b == false$
18: function LOCAL($v$):
19:     $v.b = true$
20:     return v
21:
22: $A$ = VERTEXMAP($V$, CTRUE, INIT)
23: while SIZE($U$) $\neq 0$ do
24:     EDGEMAPDENSE($V, join(E, A)$, F1, UPDATE1, COND1, R1)
25:     $B$ = VERTEXMAP($A$, COND1)
26:     $C$ = EDGEMAPSPARSE($B, E$, CTRUE, UPDATE2, COND2, R2)
27:     $A$ = VERTEXMAP(MINUS($A, C$), FILTER, LOCAL)
```

Algorithm 14 TRIANGLE COUNTING

```
function INIT($v$):
    $v.count = 0, v.out = \{\}$
    return v

function CHECK($s, d$):
    return $(s.deg > d.deg)$ or $((s.deg == d.deg)$ and $(s.id > d.id))$
function UPDATE1($s, d$):
    add $s.id$ to $d.out$
    return $d$
function R1($t, d$):
    $d.out = d.out \cup t.out$
    return $d$

function UPDATE2($s, d$):
    $d.count = d.count + intersact(s.out, d.out)$
    return $d$
function R2($t, d$):
    $d.count = d.count + t.count$
    return $d$

$U$ = VERTEXMAP($V$, CTRUE, INIT)
$U$ = EDGEMAP($U, E$, CHECK, UPDATE1, CTURE, R1)
$U$ = EDGEMAP($U, E$, CTRUE, UPDATE2, CTURE, R2)
```

line 26-27). This algorithm terminates when there are no more vertices could be added to the result.

D. Triangle Counting (TC)

Triangle Counting (TC) is a basic problem that is used as a subroutine of many important social network analysis algorithms. This application counts the number of triangles in an undirected graph, where a triangle is formed by three vertices and edges between each pair of them. The implementations of this application are provided by some vertex-centric graph frameworks. For example, there are two versions of implementations of TC in PowerGraph. The optimized one implements the "hash-table" version of "edge-iterator" algorithm described in [42].

To fit this algorithm in FLASH, we give an implementation as described by Algorithm 14, which is very simple and readable. The function CHECK is for performance consideration, and it ensures that every triangle is counted only once, instead of 3 times if it is not provided. PowerGraph needs lots of code for TC since it does not provide the serialization/deserialization semantics for users to exchange neighbor-lists. Gemini [11] does not support to implement this algorithm, since it limits the vertex properties to be fixed-length but the neighbor-lists should be maintained in this application.

E. Graph Coloring (GC)

Graph Coloring is a classic algorithmic problem in graph theory, which has been studied since the early 1970s. A graph coloring is an assignment of labels (also called "colors") to elements of a graph G subject to certain constraints. In the simplest form of this problem, it is a way of coloring the vertices of a graph such that no two adjacent vertices are of the same color; this is called a vertex coloring, which we consider in this paper.

This application is computationally hard. It is a NP-complete problem to decide if a given graph admits a k-coloring for a given k except for the cases $k \in \{0, 1, 2\}$. While in our case, we only try to find a practicable solution which uses as few colors as possible. This algorithm is based on the greedy strategy: in each step, every vertex will choose a color which is smallest and has not been used by its neighbors. The algorithm executes iteratively, until there is not any vertex changes its color in an iteration. Algorithm 14 demonstrates its implementation in FLASH.

An optimization of this algorithm is to use the asynchronous execution like that in PowerGraph [9], in which the algorithm

Algorithm 15 GRAPH COLORING

```
function INIT($v$):
    $v.c = 0, v.colors = \{\}$
    return v

function F1($s, d$):
    return $(s.deg > d.deg)$ or $((s.deg == d.deg)$ and $(s.id > d.id))$
function UPDATE1($s, d$):
    add $s.c$ to $d.colors$
    return $d$
function R1($t, d$):
    $d.colors = d.colors \cup t.colors$
    return $d$

function LOCAL1($v$):
    for $i \in \{0, 1, 2, ...\}$ do
        if $(i \notin v.colors)$ then
            $v.cc = i$
            return v

function FILTER($v$): return $v.c \neq v.cc$
function LOCAL2($v$):
    $v.c = v.cc$
    return v

$U$ = VERTEXMAP($V$, CTRUE, INIT)
while SIZE($U$) $\neq 0$ do
    $U$ = EDGEMAP($V, E$, F1, UPDATE1, CTURE, R1)
    $U$ = VERTEXMAP($V$, CTRUE, LOCAL1)
    $U$ = VERTEXMAP($V$, FILTER, LOCAL2)
```

can converge much faster. This is also the main reason that PowerGraph outperforms FLASH in some cases for this application. However, the asynchronous execution may result in more colors used, i.e., a practicable but sub-optimal solution compared to the result of a BSP-base algorithm. For the graph processing frameworks that do not support to define properties with unfixed-length on the vertices, this algorithm is not possible to be expressed directely, such as in Gemini [11] and Ligra [21].

F. K-Core Decomposition (KC)

K-core decomposition is a well-established metric which partitions a graph into layers from external to more central vertices. It is widely used in real-world applications, such as to study the clustering structure in social network analysis [43], in network visualization [44] and in bioinformatics [45]. In graph theory, a k-core of a graph G is a maximal connected sub-graph in which all vertices have the degree of at least k. Equivalently, it is one of the connected components of the subgraph of G formed by repeatedly deleting all vertices of degree less than k.

Ligra provides an algorithm for this application. In each step, it iterates over all remaining active vertices; and for each active vertex, removes it if induced degree $< k$, any vertex removed has core-number $(k-1)$ (i.e., is part of the $(k-1)$-core, but not k-core); this algorithm stops once no vertices are removed. When it finishes, vertices remaining are in the k-core. We follow this algorithm and implement it in FLASH's programming model, as Algorithm 16 shows. Besides, we further implemented an optimized algorithm in FLASH, please refer to [46] for more details of the algorithm. Although it is a shared-memory algorithm, we redesign and implement it in our model, as Algorithm 17 shows. This algorithm significantly outperforms the basic one, achieving speedups of up to two orders of magnitude.

Algorithm 16 K-CORE DECOMPOSITION

```
function INIT($v$):
    $v.d = v.deg$
    return v

function FILTER($v, k$): return $v.d < k$
function LOCAL($v, k$):
    $v.core = k - 1$
    return v

function UPDATE($s, d$):
    $d.d = d.d - 1$
    return $d$

$U$ = VERTEXMAP($V$, CTRUE, INIT)
for $k = \{1, 2, ..., n\}$ do
    while true do
        $A$ = VERTEXMAP($U$, FILTER.bind($k$), LOCAL.bind($k$))
        if (SIZE($A$) == 0) then
            break
        $U$ = MINUS($U, A$)
        EDGEMAPDENSE($A, E$, CTRUE, UPDATE, CTRUE)
    if (SIZE($U$) == 0) then
        break
```

Algorithm 17 OPTIMIZED K-CORE DECOMPOSITION

```
function INIT($v$):
    $v.core = v.deg$
    return v

function LOCAL1($v$):
    $v.cnt = 0, v.c = \{0\}$
    return v

function F1($s, d$):
    return $s.core \geq d.core$
function UPDATE1($s, d$):
    $d.cnt = d.cnt + 1$
    return $d$
function R1($t, d$):
    $d.cnt = d.cnt + t.cnt$
    return $d$

function FILTER($v$):
    return $v.cnt < v.core$

function UPDATE2($s, d$):
    $d.c[min(d.core, s.core)] = d.c[min(d.core, s.core)] + 1$
    return $d$

function LOCAL2($v$):
    $sum = 0$
    while $sum + v.c[v.core] < v.core$ do
        $sum = sum + v.c[v.core]$
        $v.core = v.core - 1$
    return v

$U$ = VERTEXMAP($V$, CTRUE, INIT)
while SIZE($U$) $\neq 0$ do
    $U$ = VERTEXMAP($V$, CTRUE, LOCAL1)
    $U$ = EDGEMAP($U, E$, F1, UPDATE1, CTRUE, R1)
    $U$ = VERTEXMAP($U$, FILTER)
    $U$ = EDGEMAPDENSE($V, join(E, U)$, CTRUE, UPDATE2, CTRUE)
    $U$ = VERTEXMAP($U$, CTRUE, LOCAL2)
```

G. Strongly Connected Components (SCC)

A directed graph is said to be strongly connected if there is a path in each direction between each pair of vertices of the graph. The strongly connected components (SCC) of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. As an important metric to measure the connectivity of a graph, it is often used as a building block in social network analysis.

It is challenging to design vertex-centric algorithms for SCC. Although there are simple sequential algorithms for computing it based on depth-first search (DFS), it cannot be applied to design parallel algorithms for computing SCC. As far as we know, Pregel+ is the only existing graph processing framework that has provided an implementation for the SCC application, thus we evaluate this implementation as a baseline in our paper. The algorithm it implemented is proposed in [47], which consists of a series of PPAs (practical Pregel algorithms). Since each individual PPA need all the necessary functions to be programmed, it is obviously not friendly to the users. Moreover, every PPA need its own implementation for parsing the input from the previous part and outputting data for the next part. Therefore, the actual lines of code for SCC are extremely high. In fact, there are 811 lines of code in total for expressing this SCC algorithm in Pregel+.

Algorithm 19 BICONNECTED COMPONENTS

```
function INIT($v$):
    $v.cid = v.id, v.d = v.deg$
    $v.dis = -1, v.p = -1, v.bcc = -1$
    return $v$

function F1($s, d$):
    return $(s.d > d.d)$ or $((s.d == d.d))$ and $(s.cid > d.cid))$
function UPDATE1($s, d$):
    $d.cid = s.cid, d.d = s.d$
    return $d$
function R1($t, d$):
    $flag = (t.d > d.d)$ or $((t.d == d.d))$ and $(t.cid > d.cid))$
    if ($flag$) then
        $d.cid = t.cid, d.d = t.d$
    return $d$

function FILTER1($v$): return $v.cid == v.id$
function LOCAL1($v$):
    $v.dis = 0$
    return $v$

function UPDATE2($s, d$):
    $d.dis = s.dis + 1$
    return $d$
function COND2($v$): return $v.dis == -1$
function R2($t, d$): return $t$

function F3($s, d$): return $s.dis == d.dis - 1$
function UPDATE3($s, d$):
    $d.p = s.id$
    return $d$
function COND3($v$): return $v.p == -1$
function R3($t, d$): return $t$

function F4($s, d$):
    return $(s.id > d.id)$ and $(d.p \neq s.id)$ and $(s.p \neq s.id)$
function JOINEDGES($V, E, f$):
    for $e \in E$ do
        $a = get(e.s),\ b = get(e.d)$
        if F4($a, b$) then
            $dsu\_union(f, a, b)$
            while $a \neq b$ do
                $da = a.dis,\ db = b.dis$
                $dp = get(a.p),\ db = get(b.p)$
                if $(da \geq db)$ then
                    if $(pa \neq pb)$ then
                        $dsu\_union(f, pa, a)$
                    $a = pa$
                if $(db \geq da)$ then
                    if $(pa \neq pb)$ then
                        $dsu\_union(f, pa, a)$
                    $b = pb$

function LOCAL3($v, f'$):
    $v.bcc = dsu\_find(f', v)$
    return $v$

$A =$ VERTEXMAP($V$, CTRUE, INIT)
while SIZE($A$) $\neq 0$ do                                    ▷ CC round
    $A =$ EDGEMAP($A, E$, F1, UPDATE1, CTRUE, R1)
$A =$ VERTEXMAP($V$, FILTER1, LOCAL1)
while SIZE($A$) $\neq 0$ do                                    ▷ BFS round
    $A =$ EDGEMAP($A, E$, CTRUE, UPDATE2, COND2, R2)
EDGEMAP($V, E$, F3, UPDATE3, COND3, R3)
$f = dsu(V)$
JOINEDGES ($V, E, f$)                                          ▷ join edges
$f' =$ REDUCE ($f$)                                            ▷ reduce
$A =$ VERTEXMAP($V$, CTRUE, LOCAL3.bind($f'$))
```

Algorithm 18 STRONGLY CONNECTED COMPONENTS

```
function INIT($v$):
    $v.scc = -1$
    return $v$

function LOCAL1($v$):
    $v.fid = v.id$
    return $v$

function F1($s, d$):
    return $s.fid < v.fid$
function M1($s, d$):
    $d.fid = min(d.fid, s.fid)$
    return $d$
function COND1($v$):
    return $v.scc == -1$
function R1($t, d$):
    $d.fid = min(d.fid, t.fid)$
    return $d$

function FILTER2($v$):
    return $v.fid == v.id$
function LOCAL2($v$):
    $v.scc = v.id$
    return $v$

function F2($s, d$):
    return $s.scc == d.fid$
function M2($s, d$):
    $d.scc = d.fid$
    return $d$
function COND2($v$):
    return $v.scc == -1$
function R2($t, d$):
    return $t$

function FILTER3($v$):
    return $v.scc == -1$

$A =$ VERTEXMAP($V$, CTRUE, INIT)
while SIZE($A$) $\neq 0$ do
                                                               ▷ phase 1
    $B =$ VERTEXMAP($A$, CTRUE, LOCAL1)
    while SIZE($B$) $\neq 0$ do
        $B =$ EDGEMAP($B, join(E, A)$, F1, M1, COND1, R1)
                                                               ▷ phase 2
    $B =$ VERTEXMAP($A$, FILTER2, LOCAL2)
    while SIZE($B$) $\neq 0$ do
        $B =$ EDGEMAP($B, join(reverse(E), A)$, F2, M2, COND2, R2)
    $A =$ VERTEXMAP($V$, FILTER3)
```

In FLASH, we implement the parallel coloring algorithm which is proposed in [48] for finding strongly connected components. It executes three phases for each superstep: in the first phase, to identify trivial SCCs; and then, in the second phase, coloring the vertices by the maximum id of the vertex that can reach it; and in the third phase, the algorithm detects one SCC for each color, by doing a traversal in the transpose of the graph and limiting the traversal to only the vertices with this color. Algorithm 18 shows the implementation of this algorithm in our framework, which is much more succinct than the algorithm in Pregel+.

H. Biconnected Components (BCC)

In graph theory, a biconnected graph is a connected and non-separable graph, meaning that if any one vertex of it is removed, the graph will remain connected. Therefore, a biconnected graph has no articulation vertices (also called cut

Algorithm 20 LABEL PROPAGATION

```
function INIT($v$):
    $v.set = \{\}$
    return v

function UPDATE1($s, d$):
    add $s.c$ to $d.set$
    return $d$
function R1($t, d$):
    $d.set = d.set \cup t.set$
    return $d$

function LOCAL1($v$):
    $max = 0$, $count = \{0\}$
    for $i \in v.set$ do
        $count[i] = count[i] + 1$
        if $(count[i] > max)$ then
            $max = count[i]$
            $v.cc = i$
    return v

function FILTER($v$):
    return $v.c \neq v.cc$
function LOCAL2($v$):
    $v.c = v.cc$, $v.set = \{\}$
    return v

$U$ = VERTEXMAP($V$, CTRUE, INIT)
for $iters \in \{1, 2, .., MaxIters\}$ do
    $U$ = EDGEMAP($V, E$, CTRUE, UPDATE1, CTRUE, R1)
    $U$ = VERTEXMAP($U$, CTRUE, LOCAL1)
    $U$ = VERTEXMAP($U$, FILTER, LOCAL2)
```

Algorithm 21 MINIMUM SPANNING FOREST

```
function F1($s, d$):
    return $s.id > d.id$
function UPDATE1($s, d, w, H$):
    add $(s, d, w)$ to $H$
    return $d$

function F2($s, d, f$):
    return $dsu\_find(f, s) \neq dsu\_find(f, d)$
function UPDATE2($s, d, w, f$):
    $dsu\_union(f, s, d)$
    return $d$

function KRUSKAL($V, E$):
    $f = dsu(V), msf = \{\}$
    sort $E$ by $w$
    for $(s, d, w) \in E$ do
        if F2($s, d, f$) then
            UPDATE2($s.d, w, f$)
            add $(s, d, w)$ to $msf$
    return $msf$

$H = \{\}$
EDGEMAPDENSE($V, E$, F1, UPDATE1.bind($H$), CTURE)
$msf$ = KRUSKAL($V, H$)
$H'$ = REDUCE($msf$)
$res$ = KRUSKAL($V, H'$)
```

vertices). The biconnected component (BCC) of a graph G is a maximal subgraph of it which is biconnected. And any connected graph can be decomposed into a tree of biconnected components. The use of biconnected components the is very important in the field of networking, because of its property of redundancy.

As same with SCC, the implementation of BCC is only available in Pregel+, while all other existing frameworks did not implement it. However, the implementation in Pregel+ is based on the algorithm proposed in [47], which is extremely complex for programming, taking more than 3000 lines of code in total.

While in FLASH, due to its high expressiveness, we could implement the algorithm for BCC as [49] proposed. This algorithm relies on an initial BFS traversal which creates a BFS tree. And an articulation vertex can be identified in the BFS tree by the fact that it has at least a single child vertex which does not have a path to any other vertex on the same BFS level as the articulation vertex that does not pass through the articulation vertex. Algorithm 19 shows the implementation of this algorithm in our framework, in which dsu_find and dsu_union are pre-defined functions provided by FLASH, to implement the disjoint set (union find algorithm) which is often used in graph applications (e.g., it is also utilized in MSF, as we will present). As we can see, this algorithm is more succinct and readable compared with the implementation in Pregel+.

I. Label Propagation (LPA)

The label propagation algorithm (LPA) is a well-known semi-supervised machine learning algorithm that assigns labels (also called classifications) to previously unlabeled vertices in the graph. At the start of the algorithm, typically, a small subset of the vertices have labels. And then, these labels are propagated iteratively to the unlabeled vertices during this algorithm [50]. The solution that it produces is not unique. It requires little prior information, and has advantages in time performance when compared with other clustering algorithms. As a consequence, the LPA algorithm is often used for finding communities in social network analysis [51].

This algorithm can be expressed in FLASH easily, with the implementation shown in Algorithm 20. We suppose that every vertex owns a label in the beginning ($v.c$), which is provided by the input data or it will be initialized as 0. In each of the following supersteps, every vertex checks all of its neighbors, and changes its label to the most frequent one among its neighbors. This algorithm terminates after executing for a number of iterations.

Similar with GC, Gemini does not support this algorithm because it limits the vertex properties to be fixed-length. PowerGraph has provided this LPA algorithm, while it is $9.3\times$ $27.1\times$ slower than FLASH. As we analyze, this performance gap comes from our efficient system implementation, although both frameworks use the same algorithm.

J. Minimum Spanning Forest (MSF)

Minimum Cost Spanning Tree/Forest (MSF) is an application that calculates a spanning tree of a connected, undirected, weighted graph. This tree should connect all the vertices of the graph with the minimum total weight of its edges. For unconnected graphs, we calculate a tree for every connected component, that is to say, to calculate a minimum spanning forest. MSF is an important graph application that is used directly in the design of networks and invoked as a subroutine in many other algorithms, such as [52], [53], [54]. As a result, sequential MSF algorithms have been well studied, for example, the Kruskal's algorithm [55]. However, distributed MSF is usually very complex. Actually, according to our survey, a very few existing graph processing frameworks have provided a distributed MSF implementation. For our evaluation, we consider Pregel+, who implemented MSF based on the parallel algorithm proposed in [56].

Thanks to the strong expressiveness of our programming model, and a series of auxiliary operators provided by FLASH, we could implement the sequential Kruskal's algorithm with few changes. The Kruskal's algorithm is typically more efficient than the algorithm of [56]. Algorithm 21 gives a brief presentation of our implementation, in which dsu_find and dsu_union are pre-defined functions provided by FLASH, to implement the disjoint set (union find algorithm) which is often used in graph applications. Obviously, it is much easier to program compared to [56], as we demonstrate by counting the LLoCs in our paper.

In the beginning of the algorithm, a minimum spanning forest is calculated inside each worker using the Kruskal's algorithm. And then the auxiliary operator REDUCE is used to reduce these local results in a new edge set. And at last, the Kruskal's algorithm is called again to get the final forest. This algorithm is correct because it can be easily proved that, if an edge is not used in the MSF of a subgraph, it is also not needed in the generation of the whole graph's MSF. In the Kruskal's algorithm, the edges have to be processed in the order of the weight. As a consequence, parallel processing is not available, so we process these edges through directly calling the condition checking function F1 and the mapping function UPDATE2, instead of using the EDGEMAP function. Although this implementation has a sacrifice in the parallelism, since this algorithm is more efficient, FLASH is still faster than Pregel+ in most cases.

M3,
R1.O4

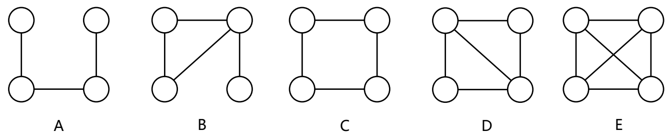


Fig. 5: Some example subgraphs (4 vertices).

K. Rectangle Counting (RC)

A rectangle in a graph is a cycle of length 4 (which consists of four vertices and four edges), as shown by Figure 5 (c). Rectangles are most elementary sub-structures in the graphs. Therefore, as same with triangle counting, rectangle counting has many important applications in the real-world, including biological network analysis, social network analysis and so on. However, in existing graph processing frameworks, none has provided an implementation for this problem. A close analysis shows that they fail to give an efficient implementation for this application since accessing the two-hop neighbors is not supported in vertex-centric frameworks.

With the programming model of FLASH, we can design an algorithm to count rectangles in a graph. Similar to the algorithm for triangle counting, it is also based on counting the intersection of two vertices' neighbor sets every time. But in this algorithm, the two sets are from two vertices that are two-hop neighbors instead of immediate neighbors. Algorithm 22 gives the implementation for this algorithm in FLASH, in which $join(E, E)$ (line 26) is the pre-defined edge set which represents two hop neighbors. By defining this custom edge set for the EDGEMAP function, this RC algorithm is expressed easily. We distinguish $v.out$ (the neighbor set of v) and $v.out_l$ (the set containing neighbors that the ids are larger than $v.id$) to ensure that each rectangle is counted only once. The sum of $v.count$ for all vertices is the total number of rectangles in the input graph G.

Algorithm 22 RECTANGLE COUNTING

```
1: function INIT($v$):
2:     $v.count = 0$
3:     $v.out = \{\}$, $v.out_l = \{\}$
4:     return v
5:
6: function UPDATE1($s, d$):
7:     if ($s.id > d.id$) then
8:         add $s.id$ to $d.out_l$
9:     add $s.id$ to $d.out$
10:    return $d$
11: function R1($t, d$):
12:    $d.out = d.out \cup t.out$
13:    $d.out_l = d.out_l \cup t.out_l$
14:    return $d$
15:
16: function F2($s.d$): return $s.id < d.id$
17: function UPDATE2($s, d$):
18:    $t = intersact(s.out_l, d.out)$
19:    $d.count = d.count + t * (t - 1)/2$
20: function R2($t, d$):
21:    $d.count = d.count + t.count$
22:    return $d$
23:
24: $U$ = VERTEXMAP($V$, CTRUE, INIT)
25: $U$ = EDGEMAP($U, E$, CTRUE, UPDATE1, CTURE, R1)
26: $U$ = EDGEMAP($U, join(E, E)$, F2, UPDATE2, CTURE, R2)
```

L. K-Clique Counting (CL)

In graph theory, a clique of an undirected graph G is a subset of vertices with each pair of vertices are adjacent. A k-clique is then a clique containing k vertices. In the k-clique counting (CL) problem, the input is an undirected graph G and a number k, and the task is to count the number of k-cliques. For our evaluation, the performance results are tested under the setting of k to be 4 (Figure 5 (e)). Since this problem is a basic metric for analyzing the topology of a graph, it has many important applications in social networks, bioinformatics, computational chemistry and other areas. However, the k-clique counting application is not provided yet in existing distributed graph processing frameworks as we surveyed.

[26] proposed a novel parallel algorithm for k-clique counting, and we port this algorithm in our model, as Algorithm 23 shows. In the beginning of the algorithm, each vertex maintains a set to record its neighbors (out), and the set $cand$ records the potential neighbors to complete the clique, which is $v.out$ initially. The counting step is completed

Algorithm 23 K-CLIQUE COUNTING

```
function INIT($v$):
    $v.count = 0, v.out = \{\}$
    return $v$

function F1($s, d$):
    return $(s.deg > d.deg)$ or $((s.deg == d.deg)$ and $(s.id > d.id))$
function UPDATE1($s, d$):
    add $s.id$ to $d.out$
    return $d$
function R1($t, d$):
    $d.out = d.out \cup t.out$
    return $d$

function FILTER($v$):
    return SIZE($v.out$) $\geq (k - 1)$

function COUNTING($cand, lev, k$):
    if ($lev == k$) then
        return SIZE($cand$)
    $t = 0$
    for $u \in cand$ do
        $cand' = intersect(cand, get(u).out)$
        if (SIZE($cand'$) $\geq k - lev - 1$) then
            $t = t+$COUNTING($cand', lev + 1, k$)
    return t

function CL($v, k$):
    $v.count$ = COUNTING($v.out, 1, k$)
    return $v$

$U$ = VERTEXMAP($V$, CTURE, INIT)
$U$ = EDGEMAP($U, E$, F1, UPDATE1, CTURE, R1)
$U$ = VERTEXMAP($U$, FILTER.bind($k$))
$U$ = VERTEXMAP($U$, CTURE, CL.bind($k$))
```

Algorithm 24 3-PATH COUNTING

```
function INIT($v$):
    $v.count = 0, v.out = \{\}$
    return v

function CHECK($s, d$):
    return $(s.deg > d.deg)$ or $((s.deg == d.deg)$ and $(s.id > d.id))$
function UPDATE1($s, d$):
    add $s.id$ to $d.out$
    return $d$
function R1($t, d$):
    $d.out = d.out \cup t.out$
    return $d$

function UPDATE2($s, d$):
    $p = intersact(s.out, d.out)$
    $d.count = d.count + (|s.out| - 1) * (|d.out| - 1) - p$
    return $d$
function R2($t, d$):
    $d.count = d.count + t.count$
    return $d$

$U$ = VERTEXMAP($V$, CTRUE, INIT)
$U$ = EDGEMAP($U, E$, CTRUE, UPDATE1, CTURE, R1)
$U$ = EDGEMAP($U, E$, CTRUE, UPDATE2, CTURE, R2)
```

through a recursive procedure. With every recursive call, a new candidate vertex u from $cand$ is added to the clique and $cand$ is pruned to contain only neighbors of u. To access the neighbors of an arbitrary vertex u, the get function which the FLASHWARE exposes is called immediately. The counts obtained from recursive calls are aggregated to get the total count for the central vertex.

M. 3-Path Counting (3PC)

A 3-path in the graph is a path containing four vertices and three edges, as Figure 5 (a) shows. To count the number of 3-paths, we can enumerate the two central vertices and then calculate all possible end-points. The computing procedure is explicated in Algorithm 24. M3, R1.O4

N. Diamond Counting (DC)

A diamond is a 4-vertex subgraph as shown by Figure 5 (d) shows. Since this kind of subgraph consists of a rectangle and an additional edge, it can be counted using an algorithm similar with RC, as shown by Algorithm 25.

Algorithm 25 DIAMOND COUNTING

```
function INIT($v$):
    $v.count = 0$
    $v.out = \{\}$
    return v

function UPDATE1($s, d$):
    add $s.id$ to $d.out$
    return $d$
function R1($t, d$):
    $d.out = d.out \cup t.out$
    return $d$

function F2($s.d$): return $(s.id < d.id) \wedge (d \in s.out)$
function UPDATE2($s, d$):
    $t = intersact(s.out, d.out)$
    $d.count = d.count + t * (t - 1)/2$
function R2($t, d$):
    $d.count = d.count + t.count$
    return $d$

$U$ = VERTEXMAP($V$, CTRUE, INIT)
$U$ = EDGEMAP($U, E$, CTRUE, UPDATE1, CTURE, R1)
$U$ = EDGEMAP($U, join(E, E)$, F2, UPDATE2, CTURE, R2)
```

Algorithm 26 TAILED TRIANGLE COUNTING

```
function INIT($v$):
    $v.count = 0, v.out = \{\}$
    return v

function CHECK($s, d$):
    return $(s.deg > d.deg)$ or $((s.deg == d.deg)$ and $(s.id > d.id))$
function UPDATE1($s, d$):
    add $s.id$ to $d.out$
    return $d$
function R1($t, d$):
    $d.out = d.out \cup t.out$
    return $d$

function UPDATE2($s, d$):
    $p = intersact(s.out, d.out)$
    $d.count = d.count + p * (|s.out| - 2) + p * (|d.out| - 2)$
    return $d$
function R2($t, d$):
    $d.count = d.count + t.count$
    return $d$

$U$ = VERTEXMAP($V$, CTRUE, INIT)
$U$ = EDGEMAP($U, E$, CTRUE, UPDATE1, CTURE, R1)
$U$ = EDGEMAP($U, E$, CTRUE, UPDATE2, CTURE, R2)
```

O. *Tailed-Triangle Counting (TTC)*

A tailed triangle is a subgraph which consists of a triangle and an additional linked vertex (Figure 5 (b)). A tailed triangle can be found by finding a triangle first and then counting all possible tailed vertices. Therefore, its algorithm is much similar with that of TC, as Algorithm 26 shows.

P. *K-Core Searching (KCS)*

In graph theory, a k-core of a graph G is a maximal connected sub-graph in which all vertices have the degree of at least k. Therefore, the k-core can be searched by repeatedly deleting all vertices of degree less than k. The algorithm in FLASH is implemented succinctly, as Algorithm 27 displays.

Algorithm 27 K-CORE SEARCHING

```
function INIT($v$):
    $v.d = v.deg$
    return v

function FILTER($v, k$): return $v.d < k$
function LOCAL($v, k$):
    $v.c = 0$
    return v

function CHECK($v, k$): return $v.d \geq k$
function F($s, d$):
    $d.c = d.c + 1$
    return $d$
function R($s, d$):
    $d.d = d.d - s.c$
    return $d$

$U$ = VERTEXMAP($V$, CTRUE, INIT)
while SIZE($U$) $\neq 0$ do
    $U$ = VERTEXMAP($U$, CTRUE, LOCAL.bind($k$))
    $U$ = VERTEXMAP($U$, FILTER.bind($k$))
    $U$ = EDGEMAPSPARSE($U, E$, CTRUE, F, CHECK.bind($k$), R)
$U$ = VERTEXMAP($V$, CHECK.bind($k$))
```